CheckboxNG Documentation

Release 0.34.0

Checkbox Developers

Jul 18, 2017

Contents

1 Installation **3**

2 Running stable release update tests **5**

3 Table of contents **7**

3.1 Introduction to Checkbox . . . 7
3.1.1 Getting Started . . . 7
3.1.2 Checkbox Command Line . . . 9
3.1.3 Looking Deeper . . . 12
3.2 Checkbox tutorials . . . 13
3.2.1 Creating an empty provider . . . 13
3.2.2 Adding a simple job to a provider . . . 13
3.2.3 Running jobs from a newly created provider . . . 13
3.2.4 Developing provider without constantly reinstalling it . . . 14
3.2.5 Improving job definition . . . 14
3.3 Checkbox Unit Types . . . 15
3.3.1 Job Unit . . . 15
3.3.2 Test Plan Unit . . . 19
3.3.3 Category Unit . . . 24
3.3.4 Resource Job Units . . . 25
3.3.5 Template Unit . . . 29
3.3.6 Exporter Unit . . . 31
3.3.7 Manifest Entry Unit . . . 33
3.3.8 Packaging Meta Data Unit . . . 35
3.3.9 Plainbox RFC822 Specification . . . 36
3.4 Reporting Bugs . . . 37
3.5 The "Checkbox Stack" . . . 37
3.5.1 Architecture Diagram . . . 37
3.5.2 Component Descriptions . . . 40
3.6 Checkbox launchers tutorial . . . 41
3.6.1 External configuration files . . . 41
3.6.2 Launcher meta-information . . . 41
3.6.3 Providers section . . . 42
3.6.4 Test plan section . . . 43
3.6.5 Test selection section . . . 43
3.6.6 User Interface section . . . 43
3.6.7 Restart section . . . 45

3.6.8 Environment section . . . 45
3.6.9 Generating reports . . . 45
3.6.10 Launcher examples . . . 47
3.7 QML-native Jobs Tutorial . . . 48
3.7.1 What is a qml-native job . . . 48
3.7.2 Software requirements . . . 48
3.7.3 First qml-native job - Smoke test . . . 50
3.7.4 How to run jobs . . . 50
3.7.5 Multi-page tests . . . 51
3.7.6 Migrating QtQuick app to a qml-native test . . . 52
3.7.7 Confined Qml jobs . . . 56
3.8 Configuration values resolution order . . . 57
3.8.1 Invoking `checkbox-cli` (without launcher) . . . 57
3.8.2 Invoking `plainbox` . . . 57
3.8.3 Invoking launcher . . . 57
3.8.4 Apps using SessionAssistant or the plainbox internals directly . . . 57
3.9 Checkbox nested test plans tutorial . . . 58
3.9.1 Quick start . . . 58
3.9.2 Use cases . . . 58
3.9.3 Known limitations . . . 63
3.10 Contributing to Snappy Testing with Checkbox . . . 64
3.10.1 Introduction . . . 64
3.10.2 Snappy Provider . . . 64
3.10.3 Creating a test in five easy steps . . . 67
3.11 Running Checkbox on Ubuntu Core . . . 72
3.11.1 Introduction . . . 72
3.11.2 Installation . . . 73
3.11.3 Running Checkbox . . . 73
3.11.4 Getting Results . . . 78
3.12 Creating a custom Checkbox application for Ubuntu Core testing . . . 80
3.12.1 Initialize the project . . . 80
3.12.2 Adding parts . . . 81
3.12.3 Create a device/project specific provider . . . 81
3.12.4 Create your new test plans (and jobs to go in them) . . . 82
3.12.5 Reusing existing provider(s) . . . 82
3.12.6 Create Checkbox Launchers configurations . . . 83
3.12.7 Create wrapper scripts . . . 83
3.12.8 Declare the launchers to be Apps that exist in your Snap . . . 84

4 Indices and tables **85**

Checkbox is a flexible test automation software. It's the main tool used in Ubuntu Certification program.

You can use checkbox without any modification to check if your system is behaving correctly or you can develop your own set of tests to check your needs. See *Checkbox tutorials* for details.

Warning: Documentation is under development. Some things are wrong, inaccurate or describe development goals rather than current state.

CHAPTER 1

Installation

Checkbox can be installed from a PPA (Personal Package Archive) (recommended) or PYPI (python package index) on Ubuntu Precise (12.04) or newer.

```
$ sudo add-apt-repository ppa:hardware-certification/public && sudo apt-get update &&␣
↪sudo apt-get install checkbox-ng
```

CHAPTER 2

Running stable release update tests

Checkbox has special support for running stable release updates tests in an automated manner. This runs all the jobs from the *sru.test plan* and sends the results to the certification website.

To run SRU tests you will need to know the so-called Secure ID of the device you are testing. Once you know that all you need to do is run:

```
$ checkbox sru $secure_id submission.xml
```

The second argument, submission.xml, is a name of the fallback file that is only created when sending the data to the certification website fails to work for any reason.

CHAPTER 3

Table of contents

Introduction to Checkbox

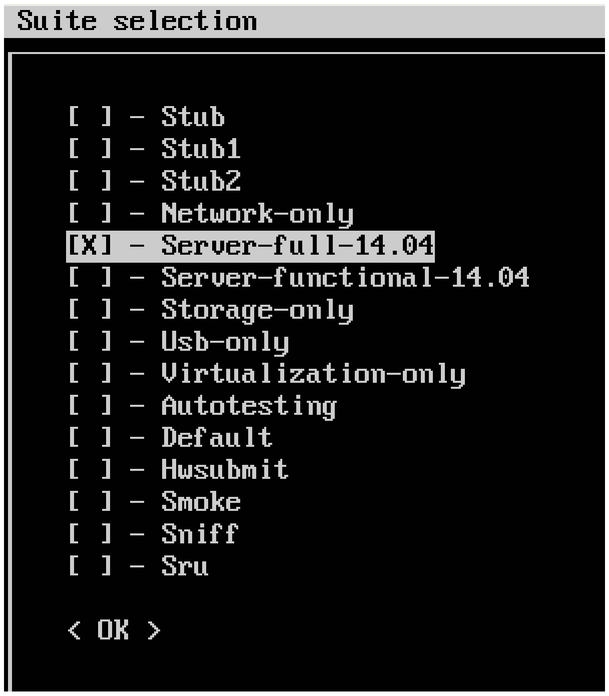
Contents

- *Introduction to Checkbox*
  - *Getting Started*
  - *Checkbox Command Line*
    - *checkbox-cli startprovider*
    - *checkbox-cli list*
    - *checkbox-cli list-bootstrapped*
    - *checkbox-cli launcher*
    - *checkbox-cli run*
  - *Looking Deeper*
    - *Providers*

Getting Started

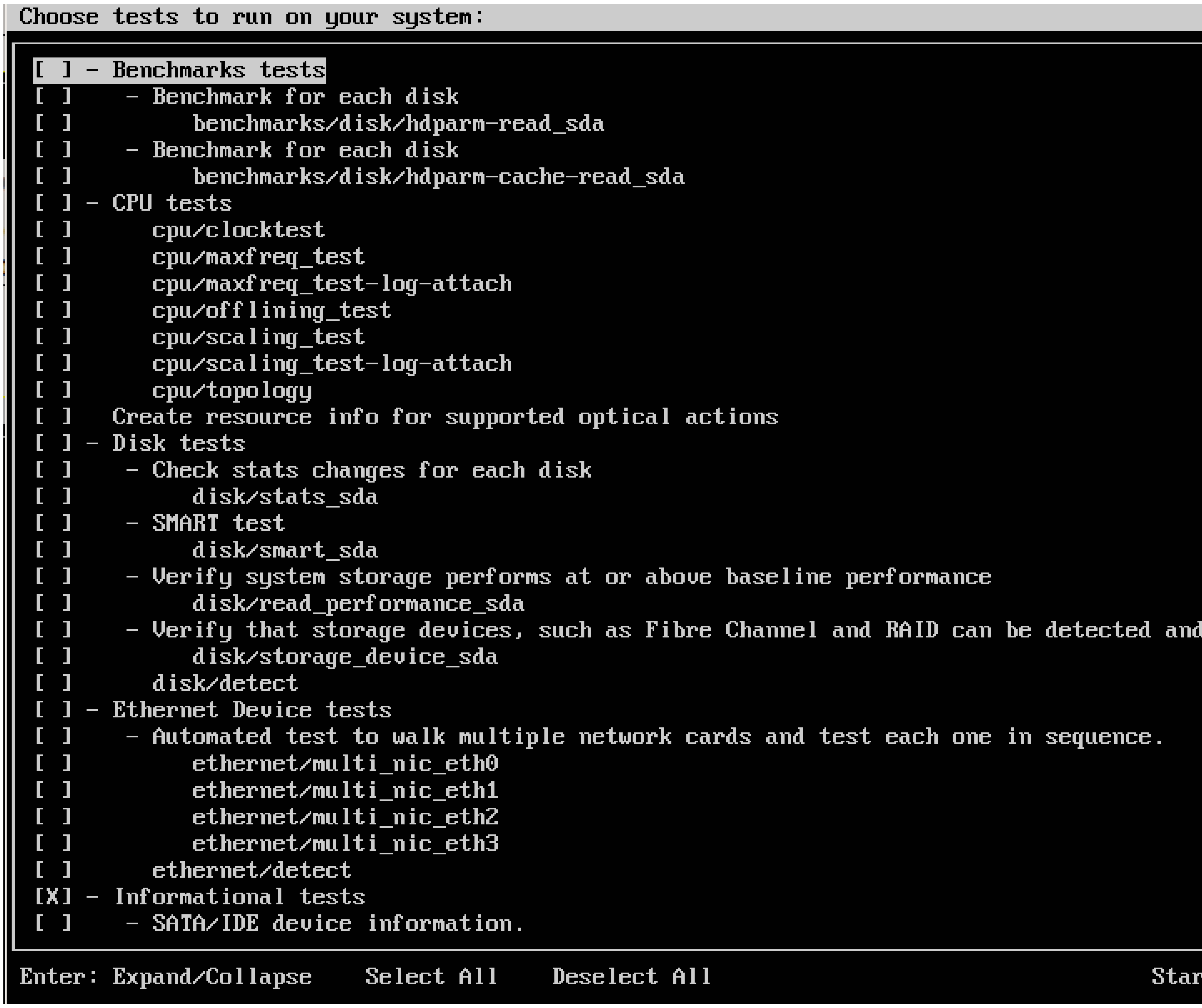
You’ve got Checkbox installed, right? *Installation*

To run command line version of Checkbox, in your terminal run `checkbox-cli`. You should be greeted with test plan selection screen:



```
Suite selection

  [ ] - Stub
  [ ] - Stub1
  [ ] - Stub2
  [ ] - Network-only
  [X] - Server-full-14.04
  [ ] - Server-functional-14.04
  [ ] - Storage-only
  [ ] - Usb-only
  [ ] - Virtualization-only
  [ ] - Autotesting
  [ ] - Default
  [ ] - Hwsubmit
  [ ] - Smoke
  [ ] - Sniff
  [ ] - Sru

  < OK >
```

With a test plan selected, you can choose the individual tests to run:



```
 Choose tests to run on your system:

  [ ] - Benchmarks tests
  [ ]    - Benchmark for each disk
  [ ]         benchmarks/disk/hdparm-read_sda
  [ ]    - Benchmark for each disk
  [ ]         benchmarks/disk/hdparm-cache-read_sda
  [ ] - CPU tests
  [ ]      cpu/clocktest
  [ ]      cpu/maxfreq_test
  [ ]      cpu/maxfreq_test-log-attach
  [ ]      cpu/offlining_test
  [ ]      cpu/scaling_test
  [ ]      cpu/scaling_test-log-attach
  [ ]      cpu/topology
  [ ]   Create resource info for supported optical actions
  [ ] - Disk tests
  [ ]    - Check stats changes for each disk
  [ ]         disk/stats_sda
  [ ]    - SMART test
  [ ]         disk/smart_sda
  [ ]    - Verify system storage performs at or above baseline performance
  [ ]         disk/read_performance_sda
  [ ]    - Verify that storage devices, such as Fibre Channel and RAID can be detected and
  [ ]         disk/storage_device_sda
  [ ]      disk/detect
  [ ] - Ethernet Device tests
  [ ]    - Automated test to walk multiple network cards and test each one in sequence.
  [ ]         ethernet/multi_nic_eth0
  [ ]         ethernet/multi_nic_eth1
  [ ]         ethernet/multi_nic_eth2
  [ ]         ethernet/multi_nic_eth3
  [ ]      ethernet/detect
  [X] - Informational tests
  [ ]    - SATA/IDE device information.

 Enter: Expand/Collapse    Select All    Deselect All                          Star
```

When the tests are run, the results are saved to files and the program prompts to submit them to Launchpad.

Checkbox Command Line

When checkbox is run without any arguments, i.e.:

```
$ checkbox-cli
```

Interactive session is started with the default options.

checkbox-cli startprovider

`startprovider` subcommand creates a new provider, e.g.:

```
$ checkbox-cli startprovider com.acme:example
```

The command will also add example units to that provider, to create an empty provider, use `--empty` option, e.g.:

```
$ checkbox-cli --empty startprovider com.acme:another-example
```

checkbox-cli list

`list` command prints out all units of the following type.

Currently there are following types you can list:

- job
- test plan
- category
- file
- template
- file
- manifest entry
- packaging meta-data
- exporter
- all-jobs (this special type list both, jobs and templates generating jobs and has a different output formatting)

Example:

```
$ checkbox-cli list job

$ checkbox-cli list "test plan"

$ checkbox-cli list all-jobs
```

Note: For multi-word types like 'test plan' remember to escape the spaces in between, or enquote the type name.

For the 'all-jobs' group, the output may be formatted to suit your needs. Use `--format` option when listing `all-jobs`. The string will be interpolated using properties of the listed jobs. Invoke `checkbox-cli list all-jobs --format ?` to see available properties. If the job definition doesn't have the specified property, `<missing $property_name>` will be printed in its place instead. Additional property - `unit_type` is provided to the formatter when listing all jobs. It is set to 'job' for normal jobs and 'template job' for jobs generated with a template unit.

Example:

```
$ checkbox-cli list all-jobs -f "{id}\n\t{tr_summary}\n"

$ checkbox-cli list all-jobs -f "{id}\n"

$ checkbox-cli list all-jobs -f "{unit_type:12} | {id:50} | {summary}\n"
```

Note: `\n` and `\t` in the formatting string are interpreted and replaced with new line and tab respectively.

When using own formatting, the jobs are not suffixed with a new line - you have to explicitly use it.

checkbox-cli list-bootstrapped

This special command list all the jobs that would be run on the device after the boostrapping phase, i.e. after all the resource jobs are run, and all of the templates were instantiated.

It requires an argument being the test plan for which the bootstrapping should execute.

Example:

```
$ checkbox-cli list-bootstrapped com.canonical.certification::default
```

checkbox-cli launcher

`launcher` command lets you customize checkbox experience.

See *Checkbox launchers tutorial* for more details.

Note: `launcher` is implied when invoking checkbox-cli with a file as the only argument. e.g.:

```
$ checkbox-cli my-launcher
```

is equivalent to:

```
$ checkbox-cli launcher my-launcher
```

checkbox-cli run

`run` lets you run particular test plan or a set of jobs.

To just run one test plan, use the test plan's id as an argument, e.g.:

```
$ checkbox-cli run com.canonical.certification::smoke
```

To run a hand-picked set of jobs, use regex pattern(s) as arguments. Jobs with id matching the expression will be run, e.g.:

```
$ checkbox-cli run com.acme:.*
```

Note: The command above runs all jobs which id begins with `com.acme:` will be run

You can use multiple patterns to match against, e.g.:

```
$ checkbox-cli run .*true .*false
```

Note: The command above runs all jobs which id ends with 'true' or 'false'

Looking Deeper

Providers

First, we installed some "provider" packages. Providers were designed to encapsulate test descriptions and their related tools and data. Providers are shipped in Debian packages, which allows us to express dependencies to ensure required external packages are installed, and we can also separate those dependencies; for instance, the provider used for server testing doesn't actually contain the server-specific test definitions (we try to keep all the test definitions in the Checkbox provider), but it does depend on all the packages needed for server testing. Most users will want the resource and Checkbox providers which contain many premade tests, but this organization allows shipping the tiny core and a fully customized provider without extraneous dependencies.

A provider is described in a configuration file (stored in `/usr/share/plainbox-providers-1`). This file describes where to find all the files from the provider. This file is usually managed automatically (more on this later). A provider can ship jobs, binaries, data and test plans.

A **job** or **test** is the smallest unit or description that Checkbox knows about. It describes a single test (historically they're called jobs). The simplest possible job is:

```
id: a-job
plugin: manual
_description: Ensure your computer is turned on. Is the computer turned on?
```

Jobs are shipped in a provider's jobs directory. This ultra-simple example has three fields: `id`, `plugin`, and `description`. (A real job should include a `_summary` field, too.) The `id` identifies the job (of course) and the `_description` provides a plain-text description of the job. In the case of this example, the description is shown to the user, who must respond because the `plugin` type is `manual`. `plugin` types include (but are not limited to):

- `manual` – A test that requires the user to perform some action and report the results.
- `shell` – An automated test that requires no user interaction; the test is passed or failed on the basis of the return value of the script or command.
- `resource` – Job that identifies the resources that the system has. (e.g. discrete GPU, Wi-Fi module). This information can later be used by other jobs to control other jobs' execution. (E.g. skip Wi-Fi tests if there's no Wi-Fi chip).
- `user-interact` – A test that asks the user to perform some action *before* the test is performed. The test then passes or fails automatically based on the output of the test. An example is `keys/media-control`, which runs a tool to detect keypresses, asks the user to press volume keys, and then exits automatically once the last key has been pressed or the user clicks the skip button in the tool.
- `user-interact-verify` – This type of test is similar to the `user-interact` test, except that the test's output is displayed for the user, who must then decide whether it has passed or failed. An example of this would be the `usb/disk_detect` test, which asks the user to insert a USB key, click the `test` button, and then verify manually that the USB key was detected correctly.
- `user-verify` – A test that the user manually performs or runs automatically and requires the user to verify the result as passed or failed. An example of this is the graphics maximum resolution test which probes the system to determine the maximum supported resolution and then asks the user to confirm that the resolution is correct.
- `qml` – A test with GUI defined in a QML file. See *QML-native Jobs Tutorial*

Checkbox tutorials

Creating an empty provider

Plainbox Providers are bundles containing information how to run tests.

To create an empty provider run:

```
$ plainbox startprovider --empty com.example:myprovider
```

`plainbox` is the internal tool of checkbox. It's used on rare occasions, like creating a new provider. `--empty` informs plainbox that you want to start from scratch. `com.example:myprovider` is the name of the provider. Providers use IQN naming, it helps in tracking down ownership of the provider.

Plainbox Jobs are the things that describe how tests are run. Those Jobs are defined in .pxu files, in 'units' directory of the provider.

The provider we've just created doesn't have that directory, let's create it:

```
$ cd com.example\:myprovider
$ mkdir units
```

Adding a simple job to a provider

Jobs loosely follow RFC822 syntax. I.e. most content follow `key:value` pattern.

Let's add a simple job that runs a command.

Open any `.pxu` file in `units` directory of the provider (if there isn't any, just create one, like `units.pxu`). And add following content:

```
id: my-first-job
flags: simple
command: mycommand
```

`id` is used for identification purposes `flags` enables extra features. In the case of `simple`, it lets us not specify all the typical fields - Checkbox will infer some values for us. `command` specifies which command to run. Here it's `mycommand`

In order for jobs to be visible in Checkbox they have to be included in some test plan. Let's add a test plan definition to the same `.pxu` file.:

```
unit: test plan
id: first-tp
name: My first test plan
include: my-first-job
```

Warning: Separated entities in the .pxu file has to be separated by at least one empty line.

Running jobs from a newly created provider

In order for Checkbox to *see* the provider we have to install it. To do so run:

```
$ sudo ./manage.py install
```

Now we're ready to launch Checkbox! Start the command line version with:

```
$ checkbox-cli
```

Follow the instructions on the screen. The test will (probably) fail, because of `mycommand` missing in your system. Let's change the job definition to do something meaningful instead. Open `units.pxu`, and change the line:

```
command: mycommand
```

to

```
command: [ `df -B 1G --output=avail $HOME |tail -n1` -gt 10 ]
```

Note: This command checks if there's at least 10GB of free space in $HOME

This change won't be available just yet, as we still have an old version of the provider installed in the system. Let's remove the previous version, and install the new one.:

```
$ sudo rm -rf /usr/local/lib/plainbox-providers-1/com.example\:myprovider/
$ sudo ./manage.py install
```

This sudo operations (hopefully) look dangerous to you. See next part to see how to avoid that.

Developing provider without constantly reinstalling it

Instead of reinstalling the provider every time you change anything in it, you can make Checkbox read it directly from the place you're changing it in.:

```
$ ./manage.py develop
```

Because now Checkbox may see two instances of the same provider, make sure you remove the previous one.

Note: `./manage.py` develop doesn't require sudo, as it makes all the references in user's home.

Improving job definition

When you run Checkbox you see the job displayed as 'my-first-job' which is the id of the job, which is not very human-friendly. This is because of the `simple` flag. Let's improve our Job definition. Open `units.pxu` and replace the job definition with:

```
id: my-first-job
_summary: 10GB available in $HOME
_description:
    this test checks if there's at least 10gb of free space in user's home
    directory
plugin: shell
estimated_duration: 0.01
command: [ `df -B 1G --output=avail $HOME |tail -n1` -gt 10 ]
```

New stuff:

```
_summary: 10GB available in $HOME
```

Summary is shown in Checkbox screens where jobs are selected. It's a human-friendly identification of the job. It should should be short (50 - 70 chars), as it's printed in one line. _ means at the beginning means the field is translatable.

```
_purpose:
    this test checks if there's at least 10gb of free space in user's home
    directory
```

Purpose as the name suggest should describe the purpose of the test.

```
plugin: shell
```

Plugin tells Checkbox what kind of job is it. `shell` means it's a automated test that runs a command and uses it's return code to determine jobs outcome.

```
estimated_duration: 0.01
```

Tells Checkbox how long the test is expected to run. This field is currently informative only.

Checkbox Unit Types

Checkbox execution is governed by Units.

All units follow *Plainbox RFC822 Specification*.

Job Unit

A job unit is a smallest unit of testing that can be performed by Checkbox. All jobs have an unique name. There are many types of jobs, some are fully automated others are fully manual. Some jobs are only an implementation detail and a part of the internal architecture of Checkbox.

File format and location

Jobs are expressed as sections in text files that conform somewhat to the `rfc822` specification format. Our variant of the format is described in rfc822. Each record defines a single job.

Job Fields

Following fields may be used by the job unit:

id: (mandatory) - A name for the job. Should be unique, an error will be generated if there are duplicates. Should contain characters in [a-z0-9/-]. This field used to be called `name`. That name is now deprecated. For backwards compatibility it is still recognized and used if `id` is missing.

summary: (mandatory) - A human readable name for the job. This value is available for translation into other languages. It is used when listing jobs. It must be one line long, ideally it should be short (50-70 characters max).

`plugin:` (mandatory) - For historical reasons it's called "plugin" but it's better thought of as describing the "type" of job. The allowed types are:

> **manual** jobs that require the user to perform an action and then decide on the test's outcome.
>
> **shell** jobs that run without user intervention and automatically set the test's outcome.
>
> **user-interact** jobs that require the user to perform an interaction, after which the outcome is automatically set.
>
> **user-interact-verify** jobs that require the user to perform an interaction, run a command after which the user is asked to decide on the test's outcome. This is essentially a manual job with a command.
>
> **attachment** jobs whose command output will be attached to the test report or submission.
>
> **resource** A job whose command output results in a set of rfc822 records, containing key/value pairs, and that can be used in other jobs' `requires` expressions.
>
> **qml** A test with GUI defined in a QML file.

`requires:` (optional). If specified, the job will only run if the conditions expressed in this field are met.

Conditions are of the form `<resource>.<key> <comparison-operator> 'value' (and|or) ...`. Comparison operators can be ==, != and `in`. Values to compare to can be scalars or (in the case of the `in` operator) arrays or tuples. The `not in` operator is explicitly unsupported.

Requirements can be logically chained with `or` and `and` operators. They can also be placed in multiple lines, respecting the rfc822 multi-line syntax, in which case all requirements must be met for the job to run (`and` ed).

`depends:` (optional). If specified, the job will only run if all the listed jobs have run and passed. Multiple job names, separated by spaces, can be specified.

`after:` (optional). If specified, the job will only run if all the listed jobs have run (regardless of the outcome). Multiple job names, separated by spaces, can be specified.

This feature is available since plainbox 0.24.

`command:` (optional). A command can be provided, to be executed under specific circumstances. For `manual`, `user-interact` and `user-verify` jobs, the command will be executed when the user presses a "test" button present in the user interface. For `shell` jobs, the command will be executed unconditionally as soon as the job is started. In both cases the exit code from the command (0 for success, !0 for failure) will be used to set the test's outcome. For `manual`, `user-interact` and `user-verify` jobs, the user can override the command's outcome. The command will be run using the default system shell. If a specific shell is needed it should be instantiated in the command. A multi-line command or shell script can be used with the usual multi-line syntax.

Note that a `shell` job without a command will do nothing.

`purpose:` (optional). Purpose field is used in tests requiring human interaction as an information about what a given test is supposed to do. User interfaces should display content of this field prior to test execution. This field may be omitted if the summary field is supplied. Note that this field is applicable only for human interaction jobs.

`steps:` (optional). Steps field depicts actions that user should perform as a part of job execution. User interfaces should display the content of this field upon starting the test. Note that this field is applicable only for jobs requiring the user to perform some actions.

`verification:` (optional). Verification field is used to inform the user how they can resolve a given job outcome. Note that this field is applicable only for jobs the result of which is determined by the user.

`user:` (optional). If specified, the job will be run as the user specified here. This is most commonly used to run jobs as the superuser (root).

`environ:` (optional). If specified, the listed environment variables (separated by spaces) will be taken from the invoking environment (i.e. the one Checkbox is run under) and set to that value on the job execution environment (i.e. the one the job will run under). Note that only the *variable names* should be listed, not the *values*, which will be taken from the existing environment. This only makes sense for jobs that also have the `user` attribute. This key provides a mechanism to account for security policies in `sudo` and `pkexec`, which provide a sanitized execution environment, with the downside that useful configuration specified in environment variables may be lost in the process.

`estimated_duration:` (optional) This field contains metadata about how long the job is expected to run for, as a positive float value indicating the estimated job duration in seconds.

Since plainbox version 0.24 this field can be expressed in two formats. The old format, a floating point number of seconds is somewhat difficult to read for larger values. To avoid mistakes test designers can use the second format with separate sections for number of hours, minutes and seconds. The format, as regular expression, is `(\d+h)?[: ]*(\d+m?)[: ]*(\d+s)?`. The regular expression expresses an optional number of hours, followed by the `h` character, followed by any number of spaces or `:` characters, followed by an optional number of minutes, followed by the `m` character, again followed by any number of spaces or `:` characters, followed by the number of seconds, ultimately followed by the `s` character.

The values can no longer be fractional (you cannot say `2.5m` you need to say `2m 30s`). We feel that sub-second granularity does is too unpredictable to be useful so that will not be supported in the future.

`flags:` (optional) This fields contains list of flags separated by spaces or commas that might induce plainbox to run the job in particular way. Currently, following flags are inspected by plainbox:

`preserve-locale:` This flag makes plainbox carry locale settings to the job's command. If this flag is not set, plainbox will neuter locale settings. Attach this flag to all job definitions with commands that use translations .

`win32:` This flag makes plainbox run jobs' commands in windows-specific manner. Attach this flag to jobs that are run on Windows OS.

`noreturn:` This flag makes plainbox suspend execution after job's command is run. This prevents scenario where plainbox continued to operate (writing session data to disk and so on), while other process kills it (leaving plainbox session in unwanted/undefined state). Attach this flag to jobs that cause killing of plainbox process during their operation. E.g. run shutdown, reboot, etc.

`explicit-fail:` Use this flag to make entering comment mandatory, when the user manually fails the job.

`has-leftovers:` This flag makes plainbox silently ignore (and not log) any files left over by the execution of the command associated with a job. This flag is useful for jobs that don't bother with maintenance of temporary directories and just want to rely on the one already created by plainbox.

`simple:` This flag makes plainbox disable certain validation advice and have some sensible defaults for automated test cases. This simplification is meant to cut the boiler plate on jobs that are closer to unit tests than elaborate manual interactions.

In practice the following changes are in effect when this flag is set:

- the *plugin* field defaults to *shell*
- the *description* field is entirely optional
- the *estimated_duration* field is entirely optional
- the *preserve-locale* flag is entirely optional

A minimal job using the simple flag looks as follows:

```
id: foo
command: echo "Jobs are simple!"
flags: simple
```

preserve-cwd: This flag makes plainbox run the job command in the current working directory without creating a temp folder (and running the command from this temp folder). Sometimes needed on snappy (See http://pad.lv/1618197)

`also-after-suspend`: See `siblings` below.

`also-after-suspend-auto`: See `siblings` below.

Additional flags may be present in job definition; they are ignored.

siblings: (optional) This field creates copies of the current job definition but using a dictionary of overridden fields. The intend is to reduce the amount of job definitions when only a few changes are required to make a job. For example we often run the same test after suspend. In that case only a new id, a new job dependency (e.g suspend/advanced) and an updated summary are required. Other possible uses of this feature are tests creation for a fixed/limited list of external ports (USB port 1, USB port 2). Useful when such enumerations cannot be computed from a resource job. This field is interpreted as a JSON blob, an array of dictionaries.

A minimal job using the siblings field looks as follows:

```
id: foo
_summary: foo foo foo
command: echo "Hello world"
flags: simple
_siblings: [
    { "id": "foo-after-suspend",
      "_summary": "foo foo foo after suspend",
      "depends": suspend/advanced}
]
```

Another example creating two more jobs in order to cover a total of 3 external USB ports:

```
id: usb_test_port1
_summary: usb stress test_(port 1)
command: usb_stress.py
flags: simple
_siblings: [
    { "id": "usb_test_port2",
      "_summary": "usb stress test_(port 2)"},
    { "id": "usb_test_port3",
      "_summary": "usb stress test_(port 3)"},
]
```

For convenience two flags can be set (`also-after-suspend` and `also-after-suspend-auto`) to create siblings with predefined settings to add "after suspend" jobs.

Given the base job:

```
id:foo
_summary: bar
flags: also-after-suspend also-after-suspend-auto
[...]
```

The `also-after-suspend` flag is a shortcut to create the following job:

```
id: after-suspend-foo
_summary: bar after suspend (S3)
depends: com.canonical.certification::suspend/suspend_advanced
```

`also-after-suspend-auto` is a shortcut to create the following job:

```
id: after-suspend-auto-foo
_summary: bar after suspend (S3)
depends: com.canonical.certification::suspend/suspend_advanced_auto
```

Warning: The curly braces used in this field have to be escaped when used in a template job (python format, Jinja2 templates do not have this issue). The syntax for templates is:

```
_siblings: [
    {{ "id": "bar-after-suspend_{interface}",
      "_summary": "bar after suspend",
      "depends": suspend/advanced}}
]
```

`imports:` (optional) This field lists all the resource jobs that will have to be imported from other namespaces. This enables jobs to use resources from other namespaces. You can use the "as ..." syntax to import jobs that have dashes, slashes or other characters that would make them invalid as identifiers and give them a correct identifier name. E.g.:

```
imports: from com.canonical.certification import cpuinfo
requires: 'armhf' in cpuinfo.platform

imports: from com.canonical.certification import cpu-01-info as cpu01
requires: 'avx2' in cpu01.other
```

The syntax of each imports line is:

```
IMPORT_STMT :: "from" <NAMESPACE> "import" <PARTIAL_ID>
             | "from" <NAMESPACE> "import" <PARTIAL_ID> AS <IDENTIFIER>
```

Test Plan Unit

The test plan unit is a facility that describes a sequence of job definitions that should be executed together.

Jobs definitions are _selected_ by either listing their identifier or a regular expression that matches their identifier. Selected jobs are executed in the sequence they appear in the list, unless they need to be reordered to satisfy dependencies which always take priority.

Test plans can contain additional meta-data which can be used in a graphical user interface. You can assign a translatable name and description to each test plan.

Test plans are also typical units so they can be defined with the familiar RFC822-like syntax that is also used for job definitions. They can also be multiple test plan definitions per file, just like with all the other units, including job definitions.

Test Plan Fields

The following fields can be used in a test plan. Note that **not all** fields need to be used or even should be used. Please remember that Checkbox needs to maintain backwards compatibility so some of the test plans it defines may have non-typical constructs required to ensure proper behavior. You don’t have to copy such constructs when working on a new test plan from scratch

`id`: Each test plan needs to have a unique identifier. This is exactly the same as with other units that have an identifier (like job definitions and categories).

This field is not used for display purposes but you may need to refer to it on command line so keeping it descriptive is useful

`name`: A human-readable name of the test plan. The name should be relatively short as it may be used to display a list of test plans to the test operator.

Remember that the user or the test operator may not always be familiar with the scope of testing that you are focusing on. Also consider that multiple test providers may be always installed at the same time. The translated version of the name (and icon, see below) is the only thing that needs to allow the test operator to pick the right test plan.

Please use short and concrete names like:

- “Storage Device Certification Tests”
- “Ubuntu Core Application’s Clock Acceptance Tests”
- “Default Ubuntu Hardware Certification Tests”.

The field has a soft limit of eighty characters. It cannot have multiple lines. This field should be marked as translatable by prepending the underscore character (_) in front. This field is mandatory.

`description`: A human-readable description of this test plan. Here you can include as many or few details as you’d like. Some applications may offer a way of viewing this data. In general it is recommended to include a description of what is being tested so that users can make an informed decision but please in mind that the `name` field alone must be sufficient to discriminate between distinct test plans so you don’t have to duplicate that information in the description.

If your tests will require any special set-up (procuring external hardware, setting some devices or software in special test mode) it is recommended to include this information here.

The field has no size limit. It can contain newline characters. This field should be marked as translatable by prepending the underscore character (_) in front. This field is optional.

`include`: A multi-line list of job identifiers or patterns matching such identifiers that should be included for execution.

This is the most important field in any test plan. It basically decides on which job definitions are selected by (included by) the test plan. Separate entries need to be placed on separate lines. White space does not separate entries as the id field may (sic!) actually include spaces.

You have two options for selecting tests:

- You can simply list the identifier (either partial or fully qualified) of the job you want to include in the test plan directly. This is very common and most test plans used by Checkbox actually look like that.
- You can use regular expressions to select many tests at the same time. This is the only way to select generated jobs (created either by template units. Please remember that the dot character has a special meaning so unless you actually want to match *any character* escape the dot with the backslash character (\).

Regardless of if you use patterns or literal job identifiers you can use their fully qualified name (the one that includes the namespace they reside in) or an abbreviated form. The abbreviated form is applicable for job definitions that reside in the same namespace (but not necessarily the same provider) as the provider that is defining the test plan.

Plainbox will catch incorrect references to unknown jobs so you should be relatively safe. Have a look at the examples section below for examples on how you can refer to jobs from other providers (you simply use their fully qualified name for that)

`mandatory_include:` A multi-line list of job identifiers or patterns matching such identifiers that should always be executed.

This optional field can be used to specify the jobs that should always run. This is particularly useful for specifying jobs that gather vital info about the tested system, as it renders impossible to generate a report with no information about system under test.

For example, session results meant to be sent to the Ubuntu certification website must include the special job: miscellanea/submission-resources

Example:

mandatory_include: miscellanea/submission-resources

Note that mandatory jobs will always be run first (along with their dependent jobs)

`bootstrap_include:` A multi-line list of job identifiers that should be run first, before the main body of testing begins. The job that should be included in the bootstrapping sections are the ones generating or helping to generate other jobs.

Example:

bootstrap_include: graphics/generator_driver_version

Note that each entry in the bootstrap_include section must be a valid job identifier and cannot be a regular expression pattern. Also note that only resource jobs are allowed in this section.

`exclude:` A multi-line list of job identifiers or patterns matching such identifiers that should be excluded from execution.

This optional field can be used to prevent some jobs from being selected for execution. It follows the similarly named `-x` command line option to the `plainbox run` command.

This field may be used when a general (broad) selection is somehow made by the `include` field and it must be trimmed down (for example, to prevent a specific dangerous job from running). It has the same syntax as the `include`.

When a job is both included and excluded, exclusion always takes priority.

`category-overrides:` A multi-line list of category override statements.

This optional field can be used to alter the natural job definition category association. Currently Plainbox allows each job definition to associate itself with at most one category (see plainbox-category-units(7) and plainbox-job-units(7) for details). This is sub-optimal as some tests can be easily assigned equally well to two categories at the same time.

For that reason, it may be necessary, in a particular test plan, to override the natural category association with one that more correctly reflects the purpose of a specific job definition in the context of a specific test plan.

For example let's consider a job definition that tests if a specific piece of hardware works correctly after a suspend-resume cycle. Let's assume that the job definition has a natural association with the category describing such hardware devices. In one test plan, this test will be associated with the hardware-specific category (using the natural association). In a special suspend-resume test plan the same job definition can be associated with a special suspend-resume category.

The actual rules as to when to use category overrides and how to assign a natural category to a specific test is not documented here. We believe that each project should come up with a workflow and semantics that best match its users.

The syntax of this field is a list of statements defined on separate lines. Each override statement has the following form:

```
apply CATEGORY-IDENTIFIER to JOB-DEFINITION-PATTERN
```

Both 'apply' and 'to' are literal strings. CATEGORY-IDENTIFIER is the identifier of a category unit. The JOB-DEFINITION-PATTERN has the same syntax as the `include` field does. That is, it can be either a simple string or a regular expression that is being compared to identifiers of all the known job definitions. The pattern can be either partially or fully qualified. That is, it may or may not include the namespace component of the job definition identifier.

Overrides are applied in order and the last applied override is the effective override in a given test plan. For example, given the following two overrides:

```
apply cat-1 to .*
apply cat-2 to foo
```

The job definition with the partial identifier `foo` will be associated with the `cat-2` category.

`estimated_duration:` An approximate time to execute this test plan, in seconds.

Since plainbox version 0.24 this field can be expressed in two formats. The old format, a floating point number of seconds is somewhat difficult to read for larger values. To avoid mistakes test designers can use the second format with separate sections for number of hours, minutes and seconds. The format, as regular expression, is `(\d+h)?[: ]*(\d+m?)[: ]*(\d+s)?`. The regular expression expresses an optional number of hours, followed by the `h` character, followed by any number of spaces or `:` characters, followed by an optional number of minutes, followed by the `m` character, again followed by any number of spaces or `:` characters, followed by the number of seconds, ultimately followed by the `s` character.

The values can no longer be fractional (you cannot say `2.5m` you need to say `2m 30s`). We feel that sub-second granularity does is too unpredictable to be useful so that will not be supported in the future.

This field is optional. If it is missing it is automatically computed by the identical field that may be specified on particular job definitions.

Since sometimes it is easier to think in terms of test plans (they are typically executed more often than a specific job definition) this estimate may be more accurate as it doesn't include the accumulated sum of mis-estimates from all of the job definitions selected by a particular test plan.

Examples

A simple test plan that selects several jobs:

```
id: foo-bar-and-froz
_name: Tests Foo, Bar and Froz
_description:
    This example test plan selects the following three jobs:
        - Foo
        - Bar
        - Froz
include:
    foo
    bar
    froz
```

A test plan that uses jobs from another provider's namespace in addition to some of its own definitions:

```
id: extended-tests
_name: Extended Storage Tests (By Corp Inc.)
_description:
    This test plan runs an extended set of storage tests, customized
    by the Corp Inc. corporation. In addition to the standard Ubuntu
    set of storage tests, this test plan includes the following tests::

    - Multipath I/O Tests
    - Degraded Array Recovery Tests
include:
    com.canonical.certification:disk/.*
    multipath-io
    degrade-array-recovery
```

A test plan that generates jobs using bootstrap_include section:

```
unit: test plan
id: test-plan-with-bootstrapping
_name: Tests with a bootstrapping stage
_description:
    This test plan uses bootstrapping_include field to generate additional
    jobs depending on the output of the generator job.
include: .*
bootstrap_include:
    generator

unit: job
id: generator
plugin: resource
_description: Job that generates Foo and Bar resources
command:
 echo "my_resource: Foo"
 echo
 echo "my_resource: Bar"

unit: template
template-unit: job
template-resource: generator
plugin: shell
estimated_duration: 1
id: generated_job_{my_resource}
command: echo {my_resource}
_description: Job instantiated from template that echoes {my_resource}
```

A test plan that marks some jobs as mandatory:

```
unit: test plan
id: test-plan-with-mandatory-jobs
_name: Test plan with mandatory jobs
_description:
    This test plan runs some jobs regardless of user selection.
include:
    Foo
mandatory_include:
    Bar

unit: job
id: Foo
```

```
_name: Foo job
_description: Job that might be deselected by the user
plugin: shell
command: echo Foo job

unit: job
id: Bar
_name: Bar job (mandatory)
_description: Job that should *always* run
plugin: shell
command: echo Bar job
```

Category Unit

The category unit is a normalized implementation of a "test category" concept. Using category units one can define logical groups of tests that deal with some specific testing area (for example, suspend-resume or USB support).

Job definitions can be associated with at most one category. Categories can be used by particular applications to facilitate test selection.

Category Fields

There are two fields that are used by the category unit:

id: This field defines the partial identifier of the category. It is similar to the id field on the job definition units.

This field is mandatory.

name: This field defines a human readable name of the category. It may be used in application user interfaces for displaying a group of tests.

This field is translatable. This field is mandatory.

Rationale

The unit is a separate entity so that it can be shipped separately of job definitions and so that it can gain a localizable name that can still be referred to uniquely by any job definition.

In the future it is likely that the unit will be extended with additional fields, for example to define an icon.

Note

Association between job definitions and categories can be overridden by a particular test plan. Please refer to the test plan unit documentation for details.

Examples

Given the following definition of a category unit:

```
unit: category
id: audio
_name: Audio tests
```

And the following definition of a job unit:

```
id: audio/speaker-headphone-plug-detection
category_id: audio
plugin: manual
_description: Plug in your headphones and ensure the system detected them
```

The job definition will be a part of the audio category.

Resource Job Units

Resources

Resources are a mechanism that allows to constrain certain job to execute only on devices with appropriate hardware or software dependencies. This mechanism allows some types of jobs to publish resource objects to an abstract namespace and to a way to evaluate a resource program to determine if a job can be started.

Resources also serve as a 'generator' for template units. See *Template Unit*

Resource Jobs

Resource Jobs are jobs with a plugin set to resource:

```
plugin: resource
```

Command that they run should print resource information in a predefined manner. This command may be considered a Resource Program

Resource programs

Resource programs are multi-line statements that can be embedded in job definitions. By far, the most common use case is to check if a required package is installed, and thus, the job can use it as a part of a test. A check like this looks like this:

```
package.name == "fwts"
```

This resource program codifies that the job needs the `fwts` package to run. There is a companion job with the same name that interrogates the local package database and publishes a set of resource objects. Each such object is a collection of arbitrary key-value pairs. The `package` job simply publishes the `name` and `version` of each installed package but the mechanism is generic and applies to all resources.

As stated, resource programs can be multi-line, a real world example of that is presented below:

```
device.category == 'CDROM'
optical_drive.cd == 'writable'
```

This example is much like the one above, referring to some resources, here coming from jobs `device` and `optical_drive`. What is important to point out is that, as a rule of a thumb, multi line programs have an implicit `and` operator between each line. This program would only evaluate to True if there is a writable CD-ROM available.

Each resource program is composed of resource expressions. Each line maps directly onto one expression so the example program above uses two resource expressions.

Resource expressions

Resource expressions are evaluated like normal python programs. They use all of the same syntax, semantics and behavior. None of the operators are overridden to do anything unexpected. The evaluator tries to follow the principle of least surprise but this is not always possible.

Resource expressions cannot execute arbitrary python code. In general almost everything is disallowed, except as noted below:

- Expressions can use any literals (strings, numbers, True, False, lists and tuples)
- Expressions can use boolean operators (`and`, `or`, `not`)
- Expressions can use all comparison operators
- Expressions can use all binary and unary operators
- Expressions can use the set membership operator (`in`)
- Expressions can use read-only attribute access

Anything else is rejected as an invalid resource expression.

In addition to that, each resource expression must use at least one variable, which must be used like an object with attributes. The name of that variable must correspond to the name of the job that generates resources. You can use the `imports` field (at a job definition level) to rename a resource job to be compatible with the identifier syntax. It can also be used to refer to resources from another namespace.

In the examples elsewhere in this page the `package` resources are generated by the `package` job. Plainbox uses this to know which resources to try but also to implicitly to express dependencies so that the `package` job does not have to be explicitly selected and marked for execution prior to the job that in fact depends on it. This is all done automatically.

Example

The job definition below generates a RTC resource:

```
id: rtc
estimated_duration: 0.02
plugin: resource
command:
  if [ -e /sys/class/rtc ]
  then
      echo "state: supported"
  else
      echo "state: unsupported"
  fi
_description: Creates resource info for RTC
```

Next let's define a Job that uses that resource.

```
plugin: shell
category_id: com.canonical.plainbox::power-management
id: power-management/rtc
requires:
  rtc.state == 'supported'
  package.name == 'util-linux'
  cpuinfo.other != 'emulated by qemu'
user: root
```

```
command: hwclock -r
estimated_duration: 0.02
_summary: Test that RTC functions properly (if present)
_description:
 Verify that the Real-time clock (RTC) device functions properly, if present.
```

Now the power-management/rtc job will only be run on systems where `/sys/class/rtc` directory exists (which is true for systems supporting RTC)

Evaluation

1. First Plainbox looks at the resource program and splits it into lines. Each non-empty line is parsed and converted to a resource expression.
2. **unexpected** Each resource expression is repeatedly evaluated, once for each resource from the group determined by the variable name. All exceptions are silently ignored and treated as if the iteration had evaluated to False. The whole resource expression evaluates to `True` if any of the iterations evaluated to `True`. In other words, there is an implicit `any()` around each resource expression, iterating over all resources.
3. **unexpected** The resource program evaluates to `True` only if all resource expressions evaluated to `True`. In other words, there is an implicit `and` between each line.

Limitations

The design of resource programs has the following shortcomings. The list is non-exhaustive, it only contains issues that we came across found not to work in practice.

Joins are not optimized

Starting with plainbox 0.24, a resource expression can use more than one resource object (resource job) at the same time. This allows the use of joins as the whole expression is evaluated over the cartesian product of all the resource records. This operation is not optimized, you can think of it as a JOIN that is performed on a database without any indices.

Let's look at a practical example:

```
package.name == desired_package.name
```

Here, two resource jobs would run. The classic *package* resource (that produces, typically, a great number of resource records, one for each package installed on the system) and a hypothetical *desired_package* resource (for this example let's pretend that it is a simple constant resource that just contains one object). Here, this operation is not any worse than before because `size(desired_package) * size(package)` is not any larger. If, however, *desired_package* was on the same order as *package* (approximately a thousand resource objects). Then the computational cost of evaluating that expression would be quadratic.

In general, the cost, assuming all resources have the same order, is exponential with the number of distinct resource jobs referenced by the expression.

Exactly one resource bound to a variable at once

It's not possible to refer to two different resources, from the same resource group, in one resource expression. In other terms, the variable always points to one object, it is not a collection of objects.

For example, let's consider this program:

```
package.name == 'xorg' and package.name == 'procps'
```

Seemingly the intent was to ensure that both `xorg` and `procps` are installed. The reason why this does not work is that at each iteration of the the expression evaluator, the name `package` refers to exactly one resource object. In other words, that expression is equivalent to this one:

```
A == True and A == False
```

This type of error is not captured by our limited semantic analyzer. It will silently evaluate to False and inhibit the job from being stated.

To work around this, split the expression to two consecutive lines. As stated in rule 3 in the list above, there is an implicit `and` operator between all expressions. A working example that expresses the same intent looks like this:

```
package.name == 'xorg'
package.name == 'procps'
```

Operator != is useless

This is strange at first but quickly becomes obvious once you recall rule 2 from the list above. That rule states that the expression is evaluated repeatedly for each resource from a particular group and that any `True` iteration marks the whole expression as `True`).

Let's look at a real-world example:

```
xinput.device_class == 'XITouchClass' and xinput.touch_mode != 'dependent'
```

So seemingly, the intent here was to have at least `xinput` resource with a `device_class` attribute equal to `XITouchClass` that has `touch_mode` attribute equal to anything but `dependent`.

Now let's assume that we have exactly two resources in the `xinput` group:

```
device_class: XITouchClass
touch_mode: dependent

device_class: XITouchClass
touch_mode: something else
```

Now, this expression will evaluate to `True`, as the second resource fulfills the requirements. Is this what the test designer had expected? That's hard to say. The problem here is that this expression can be understood as *at least one resource isn't something* **or** *all resources weren't something*. Both are equally valid desires and, depending on how the test is implemented, may or many not work correctly in practice.

Currently there is no workaround. We are considering adding a new syntax that would allow to specify this explicitly. The proposal is documented below as "implicit any(), explicit all()"

Everything is a string

Resource programs are regular python programs evaluated in unusual ways but all of the variables that are exposed through the resource object are strings.

This has considerable impact on comparison, unless you are comparing to a string the comparison will always silently fail as python has dynamic but strict, not loose types (there is no implicit type conversion). To alleviate this problem several type names / conversion functions are allowed in requirement programs. Those are:

- `int`, to convert to integer numbers
- `float`, to convert to floating point numbers
- `bool`, to convert to a boolean context

Template Unit

The template unit is a variant of Plainbox unit types. A template is a skeleton for defining additional units, typically job definitions. A template is defined as a typical RFC822-like Plainbox unit (like a typical job definition) with the exception that all the fields starting with the string `template-` are reserved for the template itself while all the other fields are a definition of all the eventual instances of the template.

Template-Specific Fields

There are four fields that are specific to the template unit:

`template-unit:` Name of the unit type this template will generate. By default job definition units are generated (as if the field was specified with the value of `job`) eventually but other values may be used as well.

This field is optional.

`template-resource:` Name of the resource job (if it is a compatible resource identifier) to use to parameterize the template. This must either be a name of a resource job available in the namespace the template unit belongs to *or* a valid resource identifier matching the definition in the `template-imports` field.

This field is mandatory.

`template-imports:` A resource import statement. It can be used to refer to arbitrary resource job by its full identifier and (optionally) give it a short variable name.

The syntax of each imports line is:

```
IMPORT_STMT ::  "from" <NAMESPACE> "import" <PARTIAL_ID>
              | "from" <NAMESPACE> "import" <PARTIAL_ID>
                AS <IDENTIFIER>
```

The short syntax exposes `PARTIAL_ID` as the variable name available within all the fields defined within the template unit. If it is not a valid variable name then the second form must be used.

This field is sometimes optional. It becomes mandatory when the resource job definition is from another provider namespace or when it is not a valid resource identifier and needs to be aliased.

`template-filter:` A resource program that limits the set of records from which template instances will be made. The syntax of this field is the same as the syntax of typical job definition unit's `requires` field, that is, it is a python expression.

When defined, the expression is evaluated once for each resource object and if it evaluates successfully to a True value then that particular resource object is used to instantiate a new unit.

This field is optional.

Instantiation

When a template is instantiated, a single record object is used to fill in the parametric values to all the applicable fields. Each field is formatted using the python formatting language. Within each field the record is exposed as the variable named by the `template_resource` field. Record data is exposed as attributes of that object.

The special parameter `__index__` can be used to iterate over the devices matching the `template-filter` field.

Examples

Basic example

The following example contains a simplified template that instantiates to a simple storage test. The test is only instantiated for devices that are considered *physical*. In this example we don't want to spam the user with a long list of loopback devices. This is implemented by exposing that data in the resource job itself:

```
id: device
plugin: resource
command:
    echo 'path: /dev/sda'
    echo 'has_media: yes'
    echo 'physical: yes'
    echo
    echo 'path: /dev/cdrom'
    echo 'has_media: no'
    echo 'physical: yes'
    echo
    echo 'path: /dev/loop0'
    echo 'has_media: yes'
    echo 'physical: no'
```

The template defines a test-storage-`XXX` test where `XXX` is replaced by the path of the device. Only devices which are *physical* according to some definition are considered for testing. This means that the record related to `/dev/loop0` will be ignored and will not instantiate a test job for that device. This feature can be coupled with the existing resource requirement to let the user know that we did see their CD-ROM device but it was not tested as there was no inserted media at the time:

```
unit: template
template-resource: device
template-filter: device.physical == 'yes'
requires: device.has_media == 'yes'
id: test-storage-{path}
plugin: shell
command: perform-testing-on --device {path}
```

Real life example

Here is a real life example of a template unit that generates a job for each hard drive available on the system:

```
unit: template
template-resource: device
template-filter: device.category == 'DISK'
plugin: shell
category_id: com.canonical.plainbox::disk
id: disk/stats_{name}
requires:
 device.path == "{path}"
 block_device.{name}_state != 'removable'
user: root
command: disk_stats_test {name}
_description: This test checks {name} disk stats, generates some activity and␣
↪rechecks stats to verify they've changed. It also verifies that disks appear in the␣
↪various files they're supposed to.
```

The `template-resource` used here (`device`) refers to a resource job using the `udev_resource` script to get information about the system. The `udev_resource` script returns a list of items with attributes such as `path` and `name`, so we can use these directly in our template.

Exporter Unit

The purpose of exporter units is to provide an easy way to customize the plainbox reports by delegating the customization bits to providers.

Each exporter unit expresses a binding between code (the entry point) and data. Data can be new options, different Jinja2 templates and/or new paths to load them.

File format and location

Exporter entry units are regular plainbox units and are contained and shipped with plainbox providers. In other words, they are just the same as job and test plan units, for example.

Fields

Following fields may be used by an exporter unit.

`id`: (mandatory) - Unique identifier of the exporter. This field is used to look up and store data so please keep it stable across the lifetime of your provider.

`summary`: (optional) - A human readable name for the exporter. This value is available for translation into other languages. It is used when listing exporters. It must be one line long, ideally it should be short (50-70 characters max).

`entry_point`: (mandatory) - This is a key for a pkg_resources entry point from the plainbox.exporters namespace. Allowed values are: jinja2, text, xlsx, json and rfc822.

`file_extension`: (mandatory) - Filename extension to use when the exporter stream is saved to a file.

`options`: (optional) - comma/space/semicolon separated list of options for this exporter entry point. Only the following options are currently supported.

text and rfc822:

- with-io-log
- squash-io-log
- flatten-io-log
- with-run-list
- with-job-list
- with-resource-map
- with-job-defs
- with-attachments
- with-comments
- with-job-via
- with-job-hash
- with-category-map

- with-certification-status

json: Same as for *text* and additionally:

- machine-json

xlsx:

- with-sys-info
- with-summary
- with-job-description
- with-text-attachments
- with-unit-categories

jinja2: No options available

data: (optional) - Extra data sent to the exporter code, to allow all kind of data types, the data field only accept valid JSON. For exporters using the jinja2 entry point, the template name and any additional paths to load files from must be defined in this field. See examples below.

Example

This is an example exporter definition:

```
unit: exporter
id: my_html
_summary: Generate my own version of the HTML report
entry_point: jinja2
file_extension: html
options:
 with-foo
 with-bar
data: {
 "template": "my_template.html",
 "extra_paths": [
     "/usr/share/javascript/lib1/",
     "/usr/share/javascript/lib2/",
     "/usr/share/javascript/lib3/"]
 }
```

The provider shipping such unit can be as follow:

```
- data
|   - my_template.css
|   - my_template.html
- units
    - my_test_plans.pxu
    - exporters.pxu
```

Note that exporters.pxu is not strictly needed to store the exporter units, but keeping them in a dedicated file is a good practice.

How to use exporter units?

In order to call an exporter unit from provider foo, you just need to use in in the launcher.

Example of a launcher using custom exporter unit:

```
#!/usr/bin/env checkbox-cli

[launcher]
launcher_version = 1

[transport:local_file]
type = file
path = /tmp/submission.html

[exporter:my_html]
unit = com.foo.bar::my_html

[report:local_html]
transport = local_file
exporter = my_html
```

For more information about generating reports see *Generating reports*

Manifest Entry Unit

A manifest entry unit describes a single entry in a *manifest* that describes the machine or device under test. The purpose of each entry is to define one specific fact. Plainbox uses such units to create a manifest that associates each entry with a value.

The values themselves can come from multiple sources, the simplest one is the test operator who can provide an answer. In more complex cases a specialized application might look up the type of the device using some identification method (such as DMI data) from a server, thus removing the extra interaction steps.

File format and location

Manifest entry units are regular plainbox units and are contained and shipped with plainbox providers. In other words, they are just the same as job and test plan units, for example.

Fields

Following fields may be used by a manifest entry unit.

id: (mandatory) - Unique identifier of the entry. This field is used to look up and store data so please keep it stable across the lifetime of your provider.

name: (mandatory) - A human readable name of the entry. This should read as in a feature matrix of a device in a store (e.g., "802.11ac wireless capability", or "Thunderbolt support", "Number of hard drive bays"). This is not a sentence, don't end it with a dot. Please capitalize the first letter. The name is used in various listings so it should be kept reasonably short.

The name is a translatable field so please prefix it with _ as in `_name:  Example`.

value-type: (mandatory) - Type of value for this entry. Currently two values are allowed: `bool` for a yes/no value and `natural` for any natural number (negative numbers are rejected).

value-units: (optional) - Units in which value is measured in. This is only used when `value-type` is equal to `natural`. For example a *"Screen size"* manifest entry could be measured in *"inch"* units.

resource-key: (optional) - Name of the resource key used to store the manifest value when representing the manifest as a resource record. This field defaults to the so-called *partial id* which is just the `id:` field as spelled in the unit definition file (so without the name space of the provider)

Example

This is an example manifest entry definition:

```
unit: manifest entry
id: has_thunderbolt
_name: Thunderbolt Support
value-type: bool
```

Naming Manifest Entries

To keep the code consistent there's one naming scheme that should be followed. Entries for boolean values must use the `has_XXX` naming scheme. This will allow us to avoid issues later on where multiple people develop manifest entries and it's all a bit weird what them mean `has_thunderbolt` or `thunderbolt_supported` or `tb` or whatever we come up with. It's a convention, please stick to it.

Using Manifest Entries in Jobs

Manifest data can be used to decide if a given test is applicable for a given device under test or not. When used as a resource they behave in a standard way, like all other resources. The only special thing is the unique name-space of the resource job as it is provided by plainbox itself. The name of the resource job is: `com.canonical.plainbox`. In practice a simple job that depends on data from the manifest can look like this:

```
unit: job
id: ...
plugin: ...
requires:
 manifest.has_thunderbolt == 'True' and manifest.ns == 'com.canonical.checkbox'
imports: from com.canonical.plainbox import manifest
```

Note that the job uses the `manifest` job from the `com.canonical.plainbox` name-space. It has to be imported using the `imports:` field as it is in a different name-space than the one the example unit is defined in (which is arbitrary). Having that resource it can then check for the `has_thunderbolt` field manifest entry in the `com.canonical.checkbox` name-space. Note that the name-space of the `manifest` job is not related to the `manifest.ns` value. Since any provider can ship additional manifest entries and then all share the flat name-space of resource attributes looking at the `.ns` attribute is a way to uniquely identify a given manifest entry.

Collecting Manifest Data

To interactively collect manifest data from a user please include this job somewhere early in your test plan: `com.canonical.plainbox::collect-manifest`.

Supplying External Manifest

The manifest file is stored in `$HOME/.local/share/plainbox/machine-manifest.json`. If the provisioning method ships a valid manifest file there it can be used for fully automatic but manifest-based deployments.

Packaging Meta Data Unit

The packaging meta-data unit describes system-level dependencies of a provider in a machine readable way. Dependencies can be specified separately for different distributions. Dependencies can also be specified for a common base distribution (e.g. for Debian rather than Ubuntu). The use of packaging meta-data units can greatly simplify management of dependencies of binary packages as it brings those decisions closer to the changes to the actual provider and makes package management largely automatic.

File format and location

Packaging meta-data units are regular plainbox units and are contained and shipped with plainbox providers. In other words, they are just the same as job and test plan units, for example.

Fields

Following fields may be used by a manifest entry unit.

`os-id`: (mandatory) - the identifier of the operating system this rule applies to. This is the same value as the `ID` field in the file `/etc/os-release`. Typical values include `debian`, `ubuntu` or `fedora`.

`os-version-id`: (optional) - the identifier of the specific version of the operating system this rule applies to. This is the same as the `VERSION_ID` field in the file `/etc/os-release`. If this field is not present then the rule applies to all versions of a given operating system.

The remaining fields are custom and depend on the packaging driver. The values for **Debian** are:

`Depends`: (optional) - a comma separated list of dependencies for the binary package. The syntax is the same as in normal Debian control files (including package version dependencies). This field can be split into multiple lines, for readability, as newlines are discarded.

`Suggests`: (optional) - same as `Depends`.

`Recommends`: (optional) - same as `Depends`.

Matching Packaging Meta-Data Units

The base Linux distribution driver parses the `/etc/os-release` file, looks at the `ID`, `ID_VERSION` and optionally the `ID_LIKE` fields. They are used as a standard way to determine the distribution for which packaging meta-data is being collected for.

The *id and version match* strategy requires that both the `os-id` and `os-dependencies` fields are present and that they match the `ID` and `ID_VERSION` values. This strategy allows the test maintainer to express each dependency accurately for each operating system they wish to support.

The *id match* strategy is only used when the `os-version` is not defined. It is useful when a single definition is applicable to many subsequent releases. This is especially useful when job works well with sufficiently old version of a third party dependency and there is no need to repeatedly re-state the same dependency for each later release of the operating system.

The *id_like match* strategy is only used as a last resort and can be seen as a weaker *id match* strategy. This time the `os-id` field is compared to the `ID_LIKE` field (if present). It is useful for working with Debian derivatives, like Ubuntu.

Each matching packaging meta-data unit is then passed to the driver to generate packaging meta-data.

Example

This is an example packaging meta-data unit, as taken from the resource provider:

```
unit: packaging meta-data
os-id: debian
Depends:
 python3-checkbox-support (>= 0.2),
 python3 (>= 3.2),
Recommends:
 dmidecode,
 dpkg (>= 1.13),
 lsb-release,
 wodim
```

This will cause the binary provider package to depend on the appropriate version of `python3-checkbox-support` and `python3` in both *Debian*, *Ubuntu* and, for example, *Elementary OS*. In addition the package will recommend some utilities that are used by some of the jobs contained in this provider.

Using Packaging Meta-Data in Debian

To make use of the packaging meta-data, follow those steps:

- Ensure that `/etc/os-release` exists in your build chroot. On Debian it is a part of the `base-files` package which is not something you have to worry about but other distributions may use different strategies.
- Mark the binary package that contains the provider with the `X-Plainbox-Provider:  yes` header.
- Add the `${plainbox:Depends}`, `${plainbox:Recommends}` and `${plainbox:Suggests}` variables to the binary package that contains the provider.
- Override the gen_control debhelper rule and run the `python3 manage.py packaging` command in addition to running `dh_gencontrol`:

  ```
  override_dh_gencontrol:
      python3 manage.py packaging
      dh_gencontrol
  ```

Plainbox RFC822 Specification

The syntax is only loosely inspired by the actual **RFC 822** syntax. Since Plainbox is not processing email, the original specification is used only as an inspiration. One of the most important aspect of the syntax we're using is relative familiarity for other users of the system and ease-of-use when using general, off-the-shelf text editors.

Backus–Naur Form

An approximated syntax can be summarized as the following BNF:

```
record-list: record-list '\n' record
             | record
record: entry-list '\n\n' entry
        | entry
entry: KEY ':' VALUE
KEY: ^[^:]+
VALUE: .+\n([ ].+)*
```

There are two quirks which not handled by this syntax (see below). Otherwise the syntax is very simple. It defines a list of records. Each record is a list of entries. Each entry is a key-value pair. Values can be multi-line, which allows for convenient expression of longer text fragments.

Quirk 1 – the magic dot

Due to the way the multi-line VALUE syntax is defined, it would be impossible (or possible but dependent only on whitespace, which is not friendly) to include two consecutive newlines. For that reason a line consisting of a single space, followed by a single dot is translated to an empty line.

The example below:

```
key:
 .
 more value
```

Is parsed as an ENTRY (in python syntax):

```
("key", "\nvalue")
```

Quirk 2 – the # comments

Since it's a line-oriented format and people are used to being able to insert comments anywhere with the `# comment` notation, any line that _starts_ with a hash or pound character is discarded. This happens earlier than other parts of parsing so comments are invisible to the rest of the parser. They can be included anywhere, including in the middle of a multi-line value.

Example:

```
# this is a comment
key: value
 multi-line
# comment!
 and more
```

Reporting Bugs

To report bugs on the Checkbox project you will need a launchpad account. You may find instructions on how to create one useful. Once you have an account you can report bugs.

The "Checkbox Stack"

The Checkbox Stack is a collection of projects that together constitute a complete testing and certification solution. It is composed of the following parts (see table below for extra details). All of the projects are linked to from the Launchpad project group.

Architecture Diagram

This diagram contains a high-level approximation of the current Checkbox architecture. There are three main "pillars". On the left we have *end products*. Those are actual tools that certification and engineers are using. On the right we have the *test market*. This is a open market of tests vendors and suppliers. The tests are wrapped in containers known as providers. In the center we have three shared components. Those implement the bulk of the framework and user interfaces for test execution. Finally in the bottom-left corner there is a part of checkbox (a library) that is shared with HEXR for certain tasks. HEXR is a out-of-scope web application used by part of the certification process. Arrows imply communication with the shape of the arrow shows who calls who.

As mentioned before, in the center column there are three main components of shared code (shared by everyone using the end products that are discussed below). The shared code is composed of plainbox, checkbox and checkbox-converged. Component responsibilities are discussed in more detail in the table below. Here we can see that checkbox and checkbox-converged use plainbox API. checkbox-converged does so using pyotherside, and checkbox uses this api directly through python 3.

In the right hand side column there are various test providers. The checkbox project is producing and maintaining a number of providers (see the table below) but it is expected that our downstream users will also produce their own providers (specific to a customer or project). Eventually some providers may come from third parties that will adopt the format.

Lastly in the bottom-left corner, the shared library, this library contains many parsers of various file formats and output formats. Technically this library is a dependency of HEXR, checkbox *and* of providers. As an added complexity the library needs to be called from python3 code and python2 code.

Note: The communication between checkbox and plainbox is bi-directional. Plainbox offers some base interfaces and extension points. Those are all exposed through plainbox (using common APIs) but some of those are actually implemented in checkbox-ng.

Warning: All internal APIs is considered unstable. Stable APIs include:

- unit definitions
- SessionAssistant API
- launcher syntax

Component Descriptions

Project	Responsible for	Type
Checkbox-Converged	• The QML user interface • The graphical launcher for providers, e.g. checkbox-certification-client	Application

Checkbox (CLI)	• The python command-line interface
	  – the text user interface
	  – the SRU testing command
	• Additional certification APIs
	  – sending data to Launchpad
	  – sending data to HEXR
		Application

Client Certification Provider	• canonical-certification-client executable • client certification test plans	Provider
Server Certification Provider	• server certification test plans • additional server test plans	Provider
System-on-Chip Server Certification Provider	• SoC server certification test plans	Provider
Checkbox Provider	• Almost all job definitions • Most of custom “scripts” • Default and SRU test plans	Provider
Resource Provider	• Almost all resource jobs • Almost all resource “scripts”	Provider
Checkbox Support	• Support code for various providers • Parsers for many text formats	Library

PlainBox	• Almost all core logic
	  – RFC822 (job definition) parser
	  – Configuration handling
	  – Testing session (suspend/resume)
	  – Job runner
	  – Trusted launcher
	  – Dependency resolver
	  – Command line handling
	  – The XML, HTML and XSLX exporters
	  – and more...
	• Provider development toolkit
	  – ‘plainbox startprovider’
	  – ‘manage.py’ implemen-
		Library and Development Toolkit

Checkbox launchers tutorial

Checkbox launchers are INI files that customize checkbox experience. The customization includes:

- choosing what jobs will be run
- how to handle machine restart
- what type of UI to use
- how to handle the results

Each section in the launcher is optional, when not supplied, the default values will be used.

This tutorial describes Launchers version 1.

External configuration files

Launcher can specify external file(s) to load values from.

`[config]`

Beginning of the configuration section.

`config_filename`

Name of the configuration file to look for. Default value: `checkbox.conf`

The directories that will be searched for the file are `/etc/xdg/` and `~/.config/`.

Example:

```
[config]
config_filename = testing.conf
```

This will make checkbox look for `/etc/xdg/testing.conf` and `~/config/testing.conf` files.

The `config_filename` may be an absolute path, and may use environment variables

Example:

```
[config]
config_filename = $MYCONFIGS/testing.conf

[config]
config_filename = /home/ubuntu/next-testing.conf
```

For more details about value resolution order see *configs*

Launcher meta-information

Launcher meta-information helps to provide consistent checkbox behaviour in the future.

`[launcher]`

Beginning of the launcher meta-information section.

`app_id`

This fields helps to differentiate between checkbox front-ends. This way sessions started with launcher with one `app_id` won't interfere with sessions started with a different launcher (provided it has `app_id` set to other value). The app_id should be in a IQN form. Default value: `com.canonical:checkbox-cli` `app_version`

This field is purely informational.

`launcher_version`

Version of the launcher language syntax and semantics to use.

`api_flags`

API flags variable determines optional feature set. List of API flags that this launcher requires. Items should be separated by spaces or commas. The default value is an empty list.

`api_version`

API version determines the behaviour of the launcher. Each checkbox feature is added at a specific API version. Default behaviours don't change silently; explicit launcher change is required. Default value: `0.99`

`stock_reports`

Stock reports are shortcuts in creating common reports. Instead of having to specify exporter, transport and a report section in a launcher, you can use any number of the stock ones. In launchers version 1 there are 4 stock reports you may use:

- `text` - print results as text on standard output
- `submission_files` - write `html`, `xlsx`, `json` and `xml` files to `$XDG_DATA_HOME` directory (or to `~/.local/share/` if `$XDG_DATA_HOME` is not defined.
- `certification` - send results to certification site
- `certification-staging` - send results to staging version of certification site

This field is a list; use commas or spaces to separate stock reports. The default value: `text, certification, submission_files`.

When using `certification` stock report, `secure_id` might be overridden by the launcher. To do this define `secure_id` in a `transport:c3` section (this is the transport that's used by the `certification` stock reports).

Launcher section example:

```
[launcher]
app_id = com.foobar:system-testing
launcher_version = 1
stock_reports = text
```

Launcher using all defaults with overridden secure_id:

```
[transport:c3]
secure_id = 001122334455667788
```

Providers section

This section provides control over which providers are used by the launcher.

`[providers]`

Beginning of the providers section.

`use`

A list of globs, from which a provider id must match at least one in order to be used. By default all providers are used.

Providers section example:

```
[providers]
use = provider1, provider2, provider-*
```

Test plan section

This section provides control over which test plans are visible in the menus and optionally forces the app to use particular one.

`[test plan]`

Beginning of the test plan section.

`unit`

An ID of a test plan that should be selected by default. By default nothing is selected.

`filter`

Glob that test plan IDs have to match in order to be visible. Default value: `*`

`forced`

If set to `yes`, test plan selection screen will be skipped. Requires `unit` field to be set. Default value: `no`.

Test selection section

This section provides lets forcing of test selection.

`[test selection]`

Beginning of the test selection section

`forced`

If set to `yes`, test selection screen will be skipped and all test specified in the test plan will be selected. Default value: `no`

User Interface section

This section controls which type of UI to use.

`[ui]`

Beginning of the user interface section

`type`

Type of UI to use. This has to be set to `interactive`, `silent`, `converged`, or `converged-silent`. Default value: `interactive`, which runs the Checkbox command line version. Note: the `converged` and `converged-silent` UI types will launch the QML interface and requires checkbox-converged to be installed on your system. Note: using `silent` or `converged-silent` UI types requires forcing test selection and test plan selection.

`dont_suppress_output`

Note: This field is deprecated, use 'output' to specify which jobs should have their output printed to the screen.

Setting this field to `yes` disables hiding of command output for jobs of type `local`, `resource` and `attachment`. Default value: `no`.

`output`

This setting lets you hide output of commands run by checkbox. It can be set to one of the following values:

- `show` - output of all jobs will be printed
- `hide-resource-and-attachment` - output of resource and attachment jobs will be hidden, output of other job types will be printed
- `hide-automated` - output of shell jobs as well as attachment and resource jobs will be hidden. Only interactive job command's output will be shown
- `hide` - same as `hide-automated`. This value is deprecated, use `hide-automated`

Default value: `show`

Note: Individual jobs can have their output hidden by specifying 'suppress-output' in their definition.

`verbosity`

This setting makes checkbox report more information from checkbox internals. Possible values are:

- `normal` - report only warnings and errors.
- `verbose` - report important events that take place during execution (E.g. adding units, starting jobs, changing the state of the session)
- `debug` - print out everything

Default value: `normal`

Note: You can also change this behavior when invoking Checkbox by using `--verbose` and `--debug` options respectively.

`auto_retry`

If set to `yes`, failed jobs will automatically be retried at the end of the testing session. In addition, the re-run screen (where user can select failed and skipped jobs to re-run) will not be shown. Default value: `no`.

`max_attempts` Defines the maximum number of times a job should be run in auto-retry mode. If the job passes, it won't be retried even if the maximum number of attempts have not been reached. Default value: `3`.

`delay_before_retry` The number of seconds to wait before retrying the failed jobs at the end of the testing session. This can be useful when the jobs relying on external factors (e.g. a WiFi access point) and you want to wait before retrying the same job. Default value: `1`.

Warning: When `auto_retry` is set to `yes`, **every** failing jobs will be retried. This can be a problem, for instance, for jobs that take a really long time to run. To avoid this, you can use the `auto-retry=no` inline override in the test plan to explicitly mark each job you do not wish to see retried.

For example:

```
id: foo-bar-and-froz
_name: Tests Foo, Bar and Froz
include:
    foo
    bar     auto-retry=no
    froz
```

In that case, even if job `bar` fails and auto-retry is activated, it will not be retried.

Restart section

This section enables fine control over how checkbox is restarted.

`[restart]`

Beginning of the restart section

`strategy`

Override the restart strategy that should be used. Currently supported strategies are `XDG` and `Snappy`. By default the best strategy is determined in runtime.

Environment section

`[environment]`

Beginning of the environment section

Each variable present in the `environment` section will be present as environment variable for all jobs run.

Example:

```
[environment]
TESTING_HOST = 192.168.0.100
```

Generating reports

Creation of reports is govern by three sections: `report`, `exporter`, and `transport`. Each of those sections might be specified multiple times to provide more than one report.

Exporter

`[exporter:exporter_name]`

Beginning of an exporter declaration. Note that `exporter_name` should be replaced with something meaningful, like `html`.

`unit`

ID of an exporter to use. To get the list of available exporter in your system run `$ plainbox dev list exporter`.

`options`

A list of options that will be supplied to the exporter. Items should be separated by spaces or commas.

Example:

```
[exporter:html]
unit = com.canonical.plainbox::html
```

Transport

`[transport:transport_name]` Beginning of a transport declaration. Note that `transport_name` should be replaced with something meaningful, like `standard_out`.

`type`

Type of a transport to use. Allowed values are: `stream`, `file`, and `certification`.

Depending on the type of transport there might be additional fields.

transport type	variables	meaning	example
`stream`	`stream`	which stream to use `stdout` or `stderr`	`[transport:out]` `type = stream` `stream = stdout`
`file`	`path`	where to save the file	`[transport:f1]` `type = file` `path = ~/report`
`certification`	`secure-id`	secure-id to use when uploading to certification sites	`[transport:c3]` `secure_id = 0123456789ABCD` `staging = yes`
	`staging`	determines if staging site should be used Default: `no`	

Report

`[report:report_name]`

Beginning of a report declaration. Note that `report_name` should be replaced with something meaningful, like `to_screen`.

`exporter`

Name of the exporter to use

`transport`

Name of the transport to use

`forced`

If set to `yes` will make checkbox always produce the report (skipping the prompt). Default value: `no`.

Example of all three sections working to produce a report:

```
[exporter:text]
unit = com.canonical.plainbox::text

[transport:out]
type = stream
stream = stdout

[report:screen]
exporter = text
transport = out
forced = yes
```

Launcher examples

1) Fully automatic run of all tests from 'com.canonical.certification::smoke' test plan concluded by producing text report to standard output.

```
#!/usr/bin/env checkbox-cli

[launcher]
launcher_version = 1
app_id = com.canonical.certification:smoke-test
stock_reports = text

[test plan]
unit = com.canonical.certification::smoke
forced = yes

[test selection]
forced = yes

[ui]
type = silent

[transport:out]
type = stream
stream = stdout

[exporter:text]
unit = com.canonical.plainbox::text

[report:screen]
transport = outfile
exporter = text
```

2) Interactive testing of FooBar project. Report should be uploaded to the staging version of certification site and saved to /tmp/submission.xml

```
#!/usr/bin/env checkbox-cli

[launcher]
launcher_version = 1
app_id = com.foobar:system-testing

[providers]
use = com.megacorp.foo::bar*

[test plan]
unit = com.megacorp.foo::bar-generic

[ui]
type = silent
output = hide

[transport:certification]
type = certification
secure-id = 00112233445566
staging = yes

[transport:local_file]
```

```
type = file
path = /tmp/submission.xml

[exporter:xml]
unit = com.canonical.plainbox::hexr

[report:c3-staging]
transport = outfile
exporter = xml

[report:file]
transport = local_file
exporter = xml
```

QML-native Jobs Tutorial

Contents

- *QML-native Jobs Tutorial*
  - *What is a qml-native job*
  - *Software requirements*
    - *Ubuntu-SDK installation*
    - *Plainbox installation*
  - *First qml-native job - Smoke test*
  - *How to run jobs*
  - *Multi-page tests*
    - *Flat page hierarchy*
    - *Using page stack*
  - *Migrating QtQuick app to a qml-native test*
    - *Plainbox job definition for the test*
    - *Testing qml job in Checkbox Touch on Ubuntu device*
  - *Confined Qml jobs*

What is a qml-native job

A qml-native job is a simple Qt Quick application (it usually is one .qml file) designed to test computer systems as any other plainbox job, difference being that it can have fully blown GUI and communicates with checkbox stack using predefined interface.

Software requirements

To develop and run qml-native jobs you need two things:

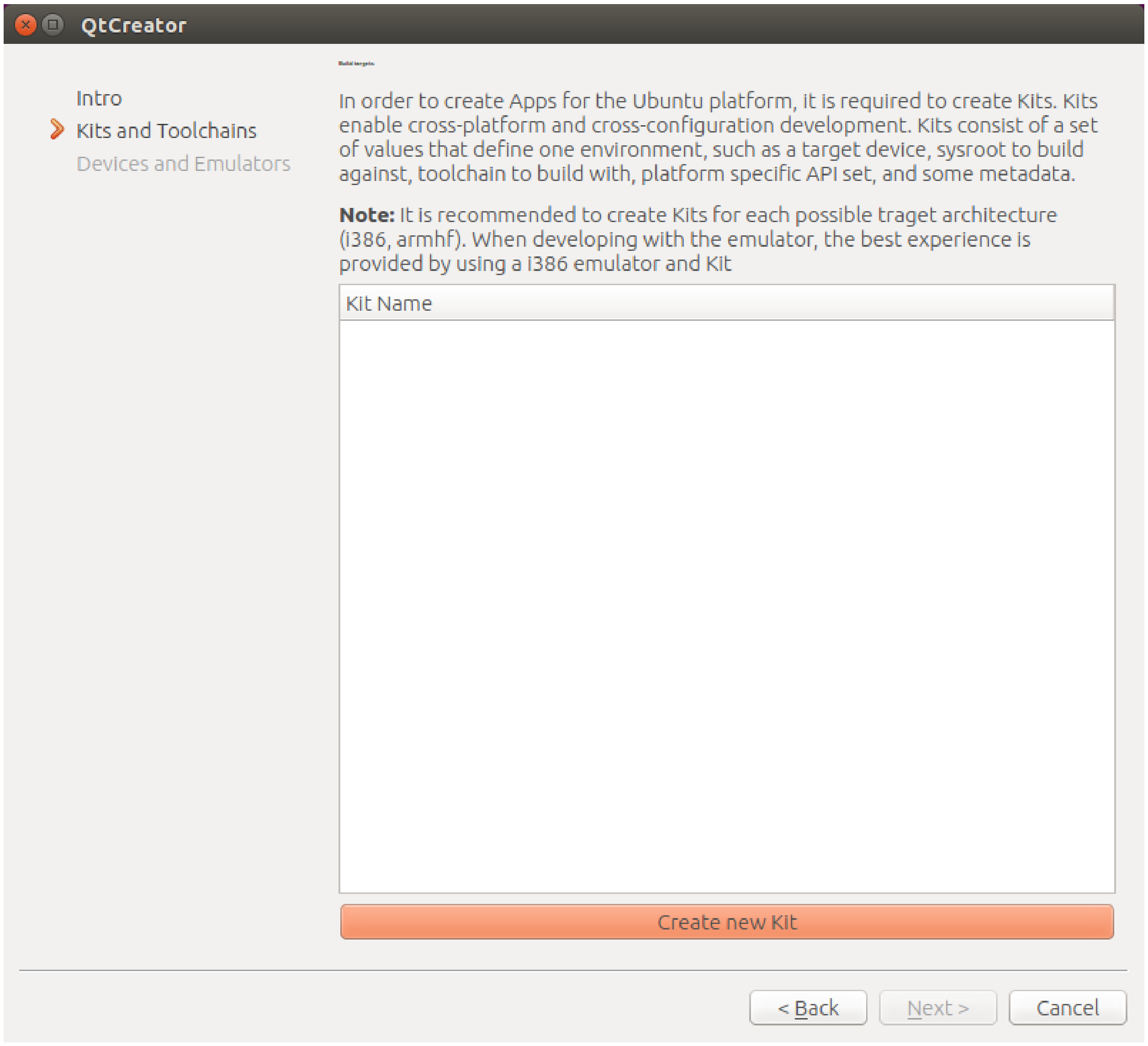
Ubuntu-SDK and Plainbox

Ubuntu-SDK installation

To install Ubuntu-SDK just run

```
# apt-get install ubuntu-sdk
```

Ubuntu-SDK, once opened, will ask you if you want to create any kit.



QtCreator

Build targets

- Intro
- Kits and Toolchains
- Devices and Emulators

In order to create Apps for the Ubuntu platform, it is required to create Kits. Kits enable cross-platform and cross-configuration development. Kits consist of a set of values that define one environment, such as a target device, sysroot to build against, toolchain to build with, platform specific API set, and some metadata.

Note: It is recommended to create Kits for each possible traget architecture (i386, armhf). When developing with the emulator, the best experience is provided by using a i386 emulator and Kit

Kit Name

Create new Kit

< Back   Next >   Cancel

Go ahead and create one matching the architecture you’re running on. And grab a coffee, as this may take awhile. If prompted about emulator installation, skip the screen.

Plainbox installation

add checkbox-dev PPA:

```
# apt-add-repository ppa:checkbox-dev/ppa
```

retrieve the list of packages:

```
# apt-get update
```

install latest plainbox

```
# apt-get install plainbox
```

If you want to work on the greatest and latest of Plainbox, you might want to use trunk version. To do that follow these steps:

```
$ bzr checkout --lightweight lp:checkbox
$ cd checkbox
$ ./mk-venv venv
$ . venv/bin/activate
```

Now you should be able to launch `plainbox-qml-shell` command.

First qml-native job - Smoke test

Let's build a very basic test that shows pass and fail buttons. All qml-native jobs start as ordinary QtQuick `Item{}`, with `testingShell` property and testDone signal. I.e.

```
import QtQuick 2.0
Item {
    property var testingShell;
     signal testDone(var test);
}
```

That's the boilerplate code every qml-native job will have. Now let's add two buttons.:

```
import QtQuick 2.0
import Ubuntu.Components 0.1
Item {
    property var testingShell;
    signal testDone(var test);
    Column {
        Button {
            text: "pass"
            onClicked: testDone({outcome: "pass"})
        }
        Button {
            text: "fail"
            onClicked: testDone({outcome: "fail"})
        }
    }
}
```

Save the above code as `simple-job.qml`. We will run it in a minute.

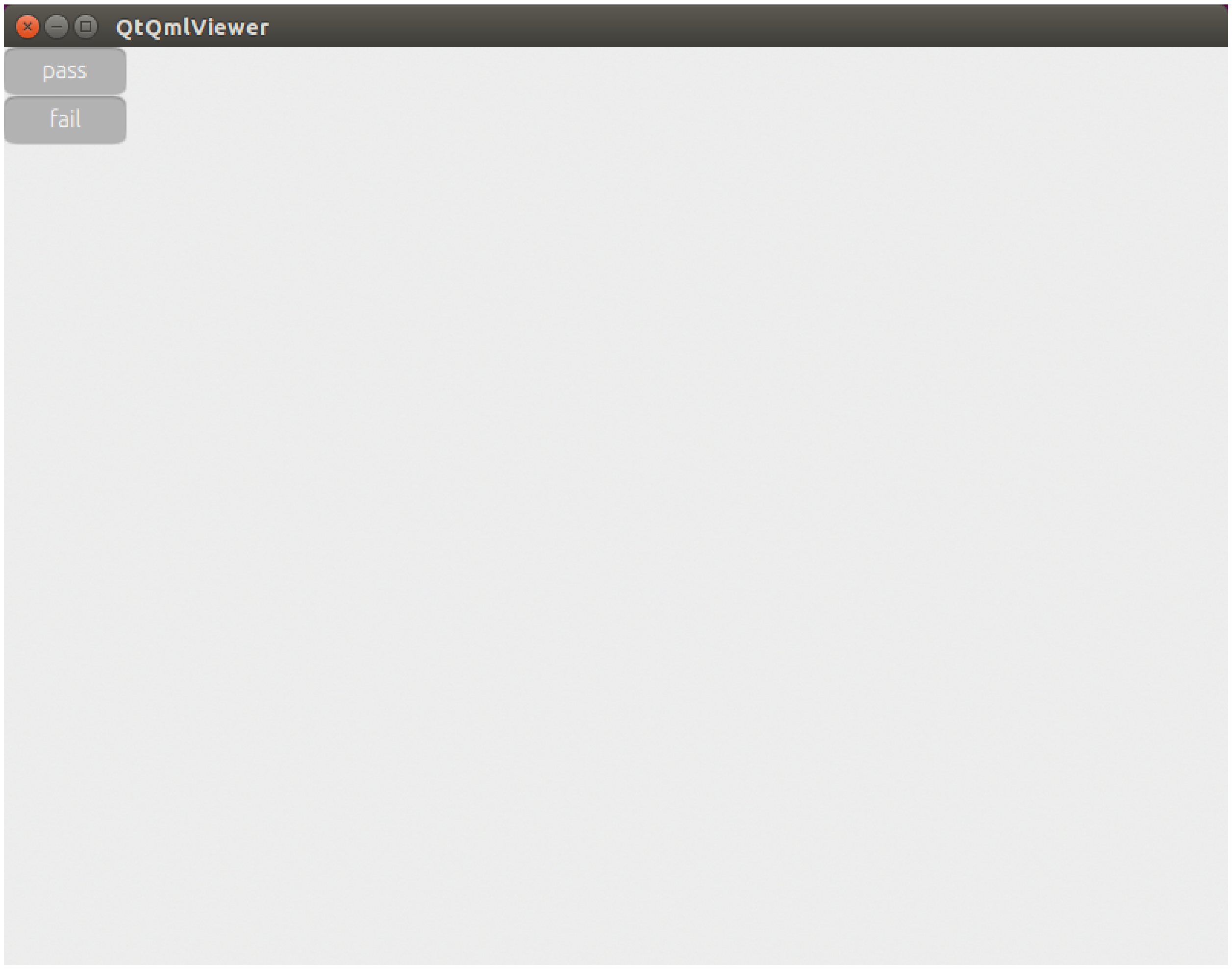
`{outcome: "pass"}` - this code creates an object with one property - `outcome` that is set the value of `"pass"`.

`testDone({outcome: "pass"})` - triggers `testDone` signal sending newly created object. This informs the governing infrastructure that the test is done and the test passed.

How to run jobs

Now we're ready to test newly developed qml job. Run:

```
$ plainbox-qml-shell simple-job.qml
```



It’s not the prettiest qml code in the world, but it is a proper qml-native plainbox job!

Multi-page tests

Two common approaches when developing multi-page qml app are flat structure, or page navigation using page stack.

Flat page hierarchy

The simplest way is to create two Page components and switch their visibility properties. E.g.:

```
Item {
    id: root
    property var testingShell;
    Page {
        id: firstPage
        Button {
            onClicked: {
                firstPage.visible = false;
                secondPage.visible = true;
            }
        }
    }
    Page {
        id: secondPage
        visible: false
    }
}
```

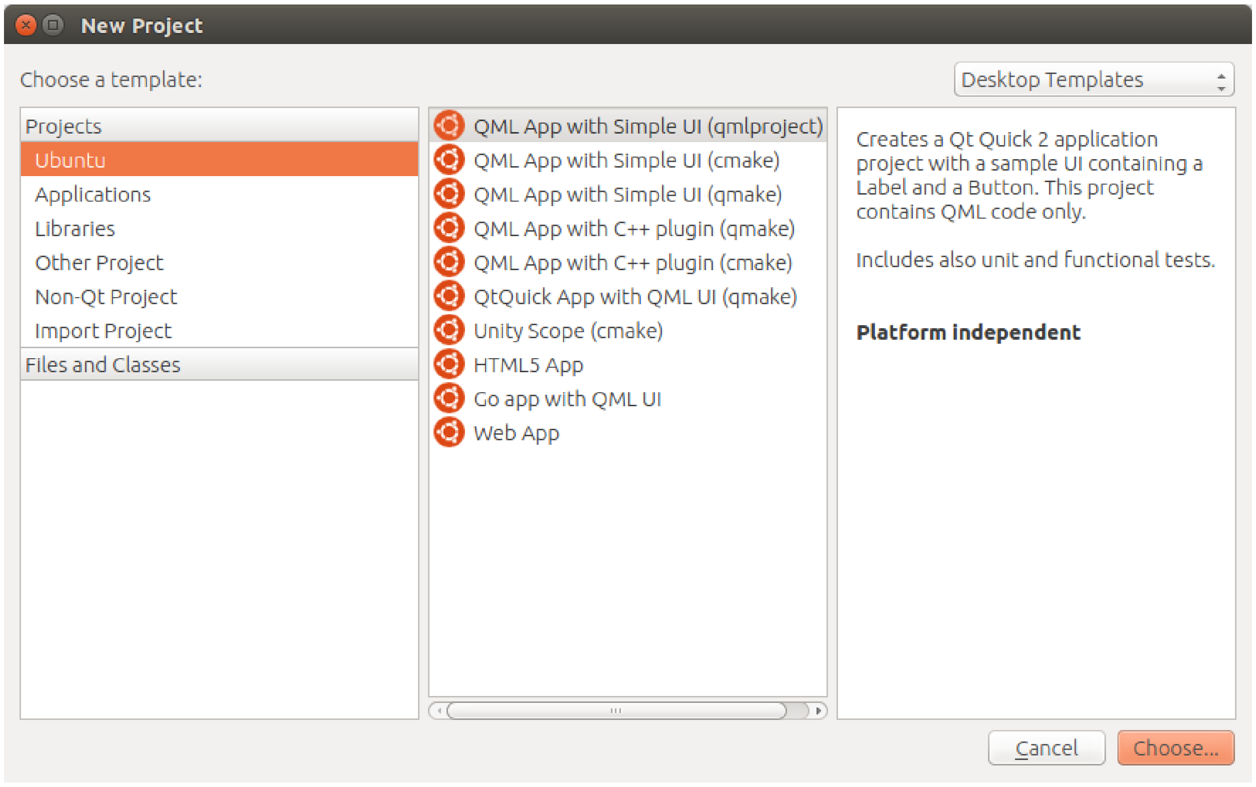
Using page stack

`testingShell` defines `pageStack` property that you can use for multi-page test with navigation. E.g.:

```
Item {
    id: root
    property var testingShell;
    Page {
        id: firstPage
        visible: false
        Button {
            onClicked: testingShell.pageStack.push(second)
        }
    }
    Page {
        id: secondPage
        visible: false
    }
    Component.onCompleted: testingShell.pageStack.push(first)
}
```

Migrating QtQuick app to a qml-native test

Start by creating ordinary “QML App with Simple UI”

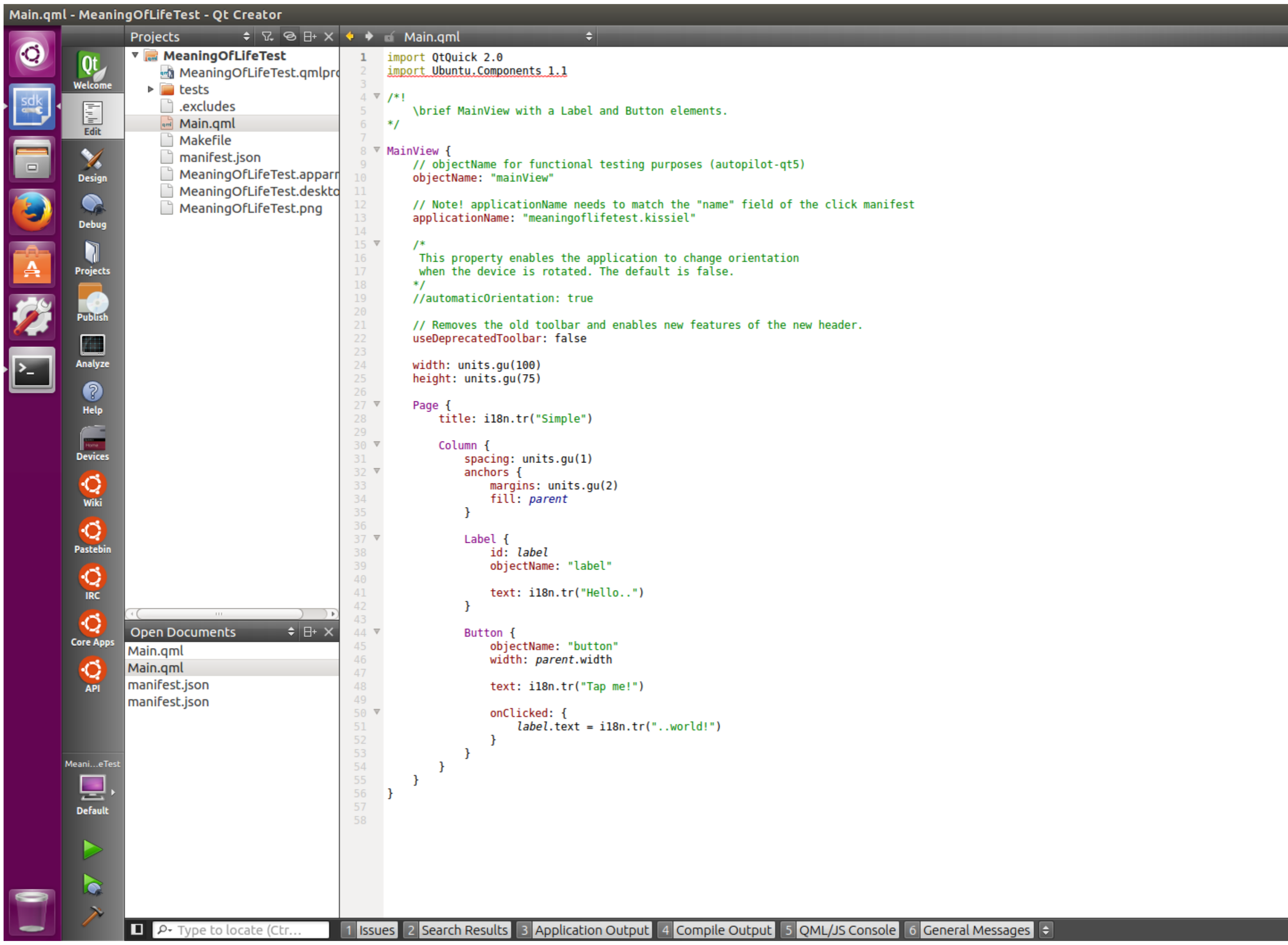


Creates a Qt Quick 2 application project with a sample UI containing a Label and a Button. This project contains QML code only.

Includes also unit and functional tests.

Platform independent

The code generated by SDK should look like this:



```
import QtQuick 2.0
import Ubuntu.Components 1.1

/*!
    \brief MainView with a Label and Button elements.
*/

MainView {
    // objectName for functional testing purposes (autopilot-qt5)
    objectName: "mainView"

    // Note! applicationName needs to match the "name" field of the click manifest
    applicationName: "meaningoflifetest.kissiel"

    /*
     This property enables the application to change orientation
     when the device is rotated. The default is false.
    */
    //automaticOrientation: true

    // Removes the old toolbar and enables new features of the new header.
    useDeprecatedToolbar: false

    width: units.gu(100)
    height: units.gu(75)

    Page {
        title: i18n.tr("Simple")

        Column {
            spacing: units.gu(1)
            anchors {
                margins: units.gu(2)
                fill: parent
            }

            Label {
                id: label
                objectName: "label"

                text: i18n.tr("Hello..")
            }

            Button {
                objectName: "button"
                width: parent.width

                text: i18n.tr("Tap me!")

                onClicked: {
                    label.text = i18n.tr("..world!")
                }
            }
        }
    }
}
```

Now you can do a typical iterative process of developing an app that should have the look and feel of the test you would like to create.

Let’s say you’re satisfied with the following app:

```
import QtQuick 2.0
import Ubuntu.Components 1.1

MainView {
    useDeprecatedToolbar: false

    width: units.gu(100)
    height: units.gu(75)

    Page {
        Column {
            spacing: units.gu(1)
            anchors {
                margins: units.gu(2)
                fill: parent
            }

            Label {
                id: label
                text: i18n.tr("4 x 7 = ?")
            }

            TextField {
                id: input
            }

            Button {
                text: i18n.tr("Check")

                onClicked: {
                    if (input.text == 28) {
                        console.log("Correct!");
                    } else {
                        console.log("Error!");
                    }
                }
            }
        }
    }
}
```

Notice that the app has a `MainView` component and one `Page` component. These are not needed in qml-native jobs, as the view is managed by the testing shell. Also, the outcome of the app is a simple `console.log()` statement. To convert this app to a proper qml-native job we need to do three things:

- remove the bits responsible for managing the view
- add `testingShell` property and the `testDone` signal
- call `testDone` once we have a result

Final result:

```
import QtQuick 2.0
import Ubuntu.Components 1.1
Item {
    property var testingShell;
    signal testDone(var test);

    Column {
        spacing: units.gu(1)
        anchors {
            margins: units.gu(2)
            fill: parent
        }

        Label {
            id: label
            text: i18n.tr("4 x 7 = ?")
        }

        TextField {
            id: input
        }

        Button {
            text: i18n.tr("Check")
            onClicked: {
                if (input.text == 28) {
                    testDone({outcome: "pass"});
                } else {
                    testDone({outcome: "fail"});
                }
            }
        }
    }
}
```

Plainbox job definition for the test

The qml file we've created cannot be considered a plainbox job until it is defined as a unit in a plainbox provider.

Consider this definition:

```
id: quazi-captcha
category_id: Captcha
plugin: qml
_summary: Basic math captcha
_description:
 This test requires user to do simple multiplication
qml_file: simple.qml
estimated_duration: 5
```

Two bits that are different in qml jobs are `plugin: qml` and `qml_file: simple.qml`

`plugin` field specifies the type of the plainbox job. The value of *qml* informs checkbox applications that this should be run in QML environment (testing shell) and `qml_file` field specifies which file serves as the entry point to the job. The file must be located in the `data` directory of the provider the job is defined in.

For other information regarding plainbox job units see:

http://plainbox.readthedocs.org/en/latest/manpages/plainbox-job-units.html

To add this job to the plainbox provider with other qml jobs, paste the job defintion to: `checkbox/providers/com.canonical.certification:qml-tests/units/qml-tests.pxu`

Testing qml job in Checkbox Touch on Ubuntu device

With job definition in qml-tests provider, and the qml file copied to its data directory we can build and install checkbox click package. In `checkbox/checkbox-touch` run:

```
./get-libs
./build-me --provider ../providers/com.canonical.certification\:qml-tests/ \
--install
```

Launch the "Checkbox" app on the device and your test should be live.

Confined Qml jobs

Sometimes there is a need to run a job with a different set of policies. Checkbox makes this possible by embedding such jobs into the resulting click package as separate apps. Each of those apps have their own apparmor declaration, so each one have its own, separate entry in the Trust database.

To request Checkbox to run a qml job as confined, add 'confined' flag to its definition.

E.g.:

```
id: confined-job
category_id: confinement-tests
plugin: qml
_summary: Job that runs as a separate app
_description:
 Checkbox should run this job with a separate set of policies.
qml_file: simple.qml
flags: confined
estimated_duration: 5
```

After the confined jobs are defined, run `generate-confinement.py` in the root directory of the provider, naming all confined jobs that have been declared.

E.g.:

```
cd my_provider
~/checkbox/checkbox-touch/confinement/generate-confinement.py confined-job
```

The tool will print all the hooks declaration you need to add to the `manifest.json` file.

Now, your multi-app click is ready to be built.

Configuration values resolution order

The directories that are searched for config files are: `/etc/xdg/` `~/.config/`

The filename that's looked up depends on how checkbox is run.

Invoking `checkbox-cli` (without launcher)

Assumed config file name is `checkbox.conf`

Invoking `plainbox`

Assumed config file name is `plainbox.conf`

Invoking launcher

The file name to look for is specified using `config_filename` variable from launcher, from the `[config]` section. If it's not present, `checkbox.conf` ' is used.

Apps using SessionAssistant or the plainbox internals directly

`plainbox.conf` is used, unless `SessionAsistant.use_alternate_configuration()` is called.

Note that if same configuration variable is defined in more then one place, the value resolution is as follows: 1. config file from `~/.config` 2. launcher being invoked (only the new syntax launchers) 3. config file from `/etc/xdg`

Checkbox nested test plans tutorial

We designed checkbox to consume test providers. Hence the test harness and the tests are completely separated. Checkbox can load tests from multiple providers. They can be installed as Debian packages or loaded from source to build a snap.

To load the tests and run them we need a test plan. Test plans for checkbox are a collection of job (test) ids meant to be run one by one.

Most of the time when we create a new test plan, there's a need to include a generic section, common to several other test plans. But the test plan unit was not allowing such feature and we ended up having a lot of duplication across our projects. And duplication means duplicated efforts to maintain all those test plan sections in sync and up-to-date.

What if it could be possible now to have nested test plans. One being built by aggregating sections from one or more "base test plans"?

Let's review in detail this new feature available in checkbox since plainbox 0.29

Quick start

The only thing to add to your test plan is the identifier of the test plan you want to include, as follow:

```
nested_part:
    com.canonical.certification::my_base_test_plan
```

The test plan order will then be test plan `include` + all nested test plan `include`, in that order.

Loading nested parts will load the `include`, `mandatory_include` and `bootstrap_include` sections and all the overrides (`category`, `certification status`).

Note: All mandatory includes will always be run first.

Note: Job and test plan ids can be listed in their abbreviated form (without the namespace prefix) if the job definitions reside in the same namespace as the provider that is defining the test plan.

Use cases

All the following examples are available here: https://github.com/yphus/nested_testplan_demo To test them locally you just need to develop the 3 providers and run one of the demo launchers:

```
git clone https://github.com/yphus/nested_testplan_demo.git
cd nested_testplan_demo/
find . -name manage.py -exec {} develop \;
./demo1 # or demo2, 3, 4, 5, 6.
```

How to use a base test plan?

Let's use two providers, both belonging to the same namespace, `com.ubuntu`:

`com.ubuntu:foo` and `com.ubuntu:baz`

Baz provider contains the following units, 4 jobs and a test plan (our base test plan):

```
id: hello
command: echo hello
flags: simple

id: bye
command: echo bye
flags: simple

id: mandatory
command: true
flags: simple

id: bootstrap
command: echo os: ubuntu
plugin: resource
flags: simple

unit: test plan
id: baz_tp
_name: Generic baz test plan
_description: This test plan contains generic test cases
estimated_duration: 1m
include:
    hello       certification-status=blocker
    bye         certification-status=non-blocker
mandatory_include:
    mandatory   certification-status=blocker
bootstrap_include:
    bootstrap
```

Foo provider contains two new tests:

```
id: always-pass
command: true
flags: simple

id: always-fail
command: true
flags: simple
```

We want to reuse the `baz_tp` in a new test plan (in the Foo provider) with the two new tests. Such test plan will look like this:

```
unit: test plan
id: foo_tp_1
_name: Foo test plan 1
_description: This test plan contains generic tests + 2 new tests
include:
    always-pass       certification-status=blocker
    always-fail
nested_part:
    baz_tp
```

The jobs execution order is:

- `bootstrap`
- `mandatory`

- `always-pass`
- `always-fail`
- `hello`
- `bye`

How to use a base test plan, but without running them last?

Let’s keep the previous providers, Foo and Baz. This time we want to run the base test plan between `always-pass` and `always-fail`. In order to change the job execution order, the new test plan will be made of several nested parts, since they will follow the list order. Let’s create in the Foo provider 2 new test plans that we’ll use as nested parts to fine tune the job ordering:

```
unit: test plan
id: foo_tp_part1
_name: Foo test plan part 1
_description: This test plan contains part 1
estimated_duration: 1m
include:
    always-pass       certification-status=blocker

unit: test plan
id: foo_tp_part2
_name: Foo test plan part 2
_description: This test plan contains part 2
estimated_duration: 1m
include:
    always-fail
```

The final test plan will only contain nested parts:

```
unit: test plan
id: foo_tp_2
_name: Foo test plan 2
_description:
 This test plan contains generic tests + 2 new tests (but ordered differently)
include:
nested_part:
    foo_tp_part1
    baz_tp
    foo_tp_part2
```

Note: Always keep the `include` section (even empty) as this field is mandatory and validation would failed otherwise (and the test plan never loaded by checkbox)

The jobs execution order is:

- `bootstrap`
- `mandatory`
- `always-pass`
- `hello`
- `bye`
- `always-fail`

How to change category or certification status of jobs coming from nested parts?

The test plan override mechanism still works with nested parts. For example the `hello` job from the Baz provider was defined as a blocker and did not have a category.

Let's update the previous use case:

```
unit: test plan
id: foo_tp_3
_name: Foo test plan 3
_description: This test plan contains generic tests + 2 new tests + overrides
include:
    always-pass       certification-status=blocker
    always-fail
nested_part:
    baz_tp
certification_status_overrides:
    apply non-blocker to hello
category_overrides:
    apply com.canonical.plainbox::audio to hello
```

To check that overrides worked as expected, you can open the json exporter report:

```
"result_map": {
    "com.ubuntu::hello": {
        "summary": "hello",
        "category_id": "com.canonical.plainbox::audio",
        "certification_status": "non-blocker"
[...]
```

How to include a nested part from another namespace?

You can include a nested part from another namespace, just prefix the test plan identifier with the provider namespace.

Let's use a third provider (Bar, under the `com.ubuntu` namespace) as an example:

```
id: sleep
command: sleep 1
flags: simple

id: uname
command: uname -a
flags: simple

unit: test plan
id: bar_tp
_name: bar test plan
_description: This test plan contains bar test cases
estimated_duration: 1m
include:
    sleep
    uname
```

Now in provider Foo, a test plan including a part from provider Bar will look like this:

```
unit: test plan
id: foo_tp_4
```

```
_name: Foo test plan 4
_description:
 This test plan contains generic tests + 2 new tests + 2 tests from a
 different namespace provider
include:
    always-pass       certification-status=blocker
    always-fail
nested_part:
    baz_tp
    com.ubuntu::bar_tp
```

The jobs execution order is:

- `bootstrap`
- `mandatory`
- `always-pass`
- `always-fail`
- `hello`
- `bye`
- `sleep`
- `uname`

Is it possible to have multiple levels of nesting?

Yes, it's possible to have multiple levels of nesting, a nested part being built from another nested parts, each level bringing its own set of new tests.

Let's add a new test plan to provider Baz:

```
unit: test plan
id: baz_tp_2
_name: Generic baz test plan 2
_description: This test plan contains generic test cases + a nested part
include:
    hello       certification-status=blocker
    bye         certification-status=non-blocker
mandatory_include:
    mandatory   certification-status=blocker
bootstrap_include:
    bootstrap
nested_part:
    com.ubuntu::bar_tp
```

As you can see this test plan includes a part from provider Bar (the same used in the previous example). In provider Foo, we can create a new test plan including *baz_tp_2*:

```
unit: test plan
id: foo_tp_5
_name: Foo test plan 5
_description: This test plan is built from multiple level of nested test plans
include:
    always-pass       certification-status=blocker
    always-fail
```

```
nested_part:
    baz_tp_2
```

The jobs execution order is still:

- `bootstrap`
- `mandatory`
- `always-pass`
- `always-fail`
- `hello`
- `bye`
- `sleep`
- `uname`

How to use a base test plan except a few jobs?

The test plan units support an optional field - `exclude` - that we can use to remove jobs from a nested part `include` section.

Note: The `exclude` ids cannot remove jobs that are parts of the `mandatory_include` sections (nested or not).

The test plan below (from provider Foo) won't run the `hello` job of provider Baz:

```
unit: test plan
id: foo_tp_6
_name: Foo test plan 6
_description: This test plan contains generic tests + 2 new tests - hello job
include:
    always-pass       certification-status=blocker
    always-fail
exclude:
    hello
nested_part:
    baz_tp
```

The jobs execution order is:

- `bootstrap`
- `mandatory`
- `always-pass`
- `always-fail`
- `bye`

Known limitations

You can create infinite loops if a nested part is calling itself or if somewhere in the nested chain such a loop exists. Checkbox won't like that and so far there's no validation to prevent it, be warned!

Contributing to Snappy Testing with Checkbox

Introduction

To support the release of devices running snappy Ubuntu Core, Canonical has produced versions of Checkbox tailored specifically for these systems.

This document aims to provide the reader with enough information to contribute new tests, or modify existing tests, with the goal of increasing coverage wherever possible.

Brief anatomy of a Checkbox test tool

Checkbox test tools consist of a number components falling into three categories:

- Core testing framework (known as Plainbox)
- UI and launchers
- Test definitions and associated data contained in a "Provider"

To add tests one need only know the specifics of the Provider(s) that form their test tool. The rest of this document will focus on Checkbox Providers and how to work on them.

Snappy Provider

The Provider housing the majority of tests for snappy Ubuntu Core systems is known as plainbox-provider-snappy and can be found in this launchpad project: https://launchpad.net/plainbox-provider-snappy

All the code both for the core of Checkbox itself and for the tests is also hosted on Launchpad. Refer to the instructions on the Code subpage to retrieve the source files for the provider: https://code.launchpad.net/plainbox-provider-snappy

Directory structure of the Provider

Using git to clone the provider, described above, will result in a directory that looks like this (at time of writing):

```
checkbox@xenial:~$ ls -1 plainbox-provider-snappy/
plainbox-provider-snappy
plainbox-provider-snappy-resource
```

The first directory listed is the provider holding the tests, the second is a supporting provider which gathers information about the system at the start of a test run. Lets look in more detail at the test provider:

```
checkbox@xenial:~$ ls -1 plainbox-provider-snappy/plainbox-provider-snappy
bin
data
manage.py
po
src
units
```

bin	Executable scripts that can be called as part of the test (refer to command field below)
data	Data to support the running of tests e.g. configuration files
manage.py	Provider management script. Must be present in each provider to specify unique identifiers.
po	Translation support, files here are used to provide translations for tests fields in to other languages.
src	Source files and accompanying build scripts e.g. C source code and a Makefile, that are compiled in to binaries and packaged with the provider for use as part of the test (refer to command field below)
units	"Job" definition files

Jobs

A Job is Checkbox parlance for an individual test. They are defined in text files whose syntax is loosely based on RFC 822. Here is an example from plainbox-provider-snappy:

```
id: cpu/offlining_test
_summary:
 Test offlining of each CPU core
_description:
 Attempts to offline each core in a multicore system.
plugin: shell
command: cpu_offlining
category_id: cpu
estimated_duration: 1s
user: root
```

An overview of the fields in this example test:

id	A unique identifier for the job
summary	A human readable name for the job. It must be one line long, ideally it should be short (50-70 characters max)
plugin	Best thought of as describing the “type” of job. Note that it is preferred for jobs to automated wherever possible so as to minimize both time to complete and possibility for operator error. The key job types starting with the most automated are: • shell - Run the command field and use the return value to determine the test result • user-interact - Ask the user to perform an action and then run the command field and use the return value to determine the test result • user-interact-verify - Ask the user to perform an action, then run the command field, and then ask the user to determine the test result .g. by examining the command output or observing some physical behaviour • manual - The last resort, just asks the user to both carry out some action(s) and then determine the test result
command	A command or script to run as part of the test. A multi-line command or shell script can be used. Refer to the plugin field above for significance to the test outcome.
category_id	Groups tests together for convenience in UIs etc.
estimated_duration	An estimate of the time taken to execute the job. Uses hours(h), minutes(m) and seconds(s) format e.g. 1h 23m 4s

Further reading: http://plainbox.readthedocs.org/en/latest/manpages/plainbox-job-units.html

Test plans

Test Plans are a facility for describing a sequence of Job definitions that should be executed together. Jobs definitions are selected for inclusion in a Test Plan by either listing their identifier (see id: field above) or by inclusion of a regular expression that matches their identifier.

Here is an example of a Test Plan from plainbox-provider-snappy, it has been abbreviated:

```
id: snappy-generic
unit: test plan
_name: QA tests for Snappy Ubuntu Core devices
estimated_duration: 1h
include:
 wifi/.*
 audio/.*
```

id	A unique identifier for the test plan
unit	Distinguishes this definition from that of e.g. a test
_name	A human readable name for the test plan
estimated_duration	A estimate of the time taken to execute the test plan. Uses hours(h), minutes(m) and seconds(s) format e.g. 1h 23m 4s
include _id	The list of tests that make up the test plan. It can be multi-line and include individual job identifiers or patterns matching multiple identifiers

Further reading: http://plainbox.readthedocs.org/en/latest/manpages/plainbox-test-plan-units.html

Creating a test in five easy steps

1. Configure your development environment

Development of Checkbox tests is best carried out on an Ubuntu Desktop system. You will need either a dedicated PC or Virtual Machine running Ubuntu Desktop 16.04 (Xenial Xerus) to gain access to the tools supporting the building of packages for snappy Ubuntu Core.

When your system is up and running make sure the following packages are installed:

```
$ sudo apt install snapcraft git:
```

And to ease development, remove these pre-installed providers:

```
$ sudo apt remove plainbox-provider-checkbox plainbox-provider-resource-generic

  You should now have all the tools required to modify and build a provider.
```

2. Get the source

Clone the providers:

```
$ git clone https://git.launchpad.net/plainbox-provider-snappy
```

Clone the snapcraft packaging branch:

```
$ git clone https://git.launchpad.net/~checkbox-dev/plainbox-provider-snappy/+git/
↪packaging
```

Further instructions will assume these were cloned in to your user's home directory.

3. Make your changes

The units folder contains a number of files named after categories. This is not a requirement, but has been used here too make finding tests a bit easier. Either create a new file or edit an existing category.:

```
$ git checkout -b <NEW-BRANCH>
$ touch ~/plainbox-provider-snappy/plainbox-provider-snappy/units/<category>.pxu
$ editor ~/plainbox-provider-snappy/plainbox-provider-snappy/units/<category>.pxu
```

If adding a new test, make sure to add the test id to the "includes" section of any test plans you'd like this test to be part of.

4. Check your test is valid

Use the provider management script to check the provider is still valid after your modifications:

```
$ cd ~/plainbox-provider-snappy/plainbox-provider-snappy-resource
$ ./manage.py develop
$ cd ~/plainbox-provider-snappy/plainbox-provider-snappy
$ ./manage.py validate
```

The validate tool will provide advisories to indicate places where you provider does not follow best practices, warnings to indicate places where runtime issues could arise, and errors to indicate things which must be fixed for the provider to be parsed and run correctly by Checkbox. This validation result is given in the last line:

```
The provider seems to be valid
```

5. Build the Checkbox snap package

The tools to build a new version of the Checkbox tool snap package are found in your clone of the packaging branch. This uses the snapcraft tool which is controlled by the snapcraft.yaml file. To build a snap with your local changes examine this file for the source sections of the provider parts:

```
$ editor ~/packaging/snapcraft.yaml

...
    plainbox-provider-snappy:
        after: [checkbox]
...
```

Modify these so the point to your local providers:[a][b]:

```
...
    plainbox-provider-snappy:
        source: <path-to-local-provider>
        source-type: local
        after: [checkbox]
...
```

Then you can build the snap package:

```
$ snapcraft clean
...
$ snapcraft
...
Snapped checkbox-snappy_0.10~s16_amd64.snap
```

6. Run the tests

See *Running Checkbox on Ubuntu Core* which describes the process of installing and running the snap.

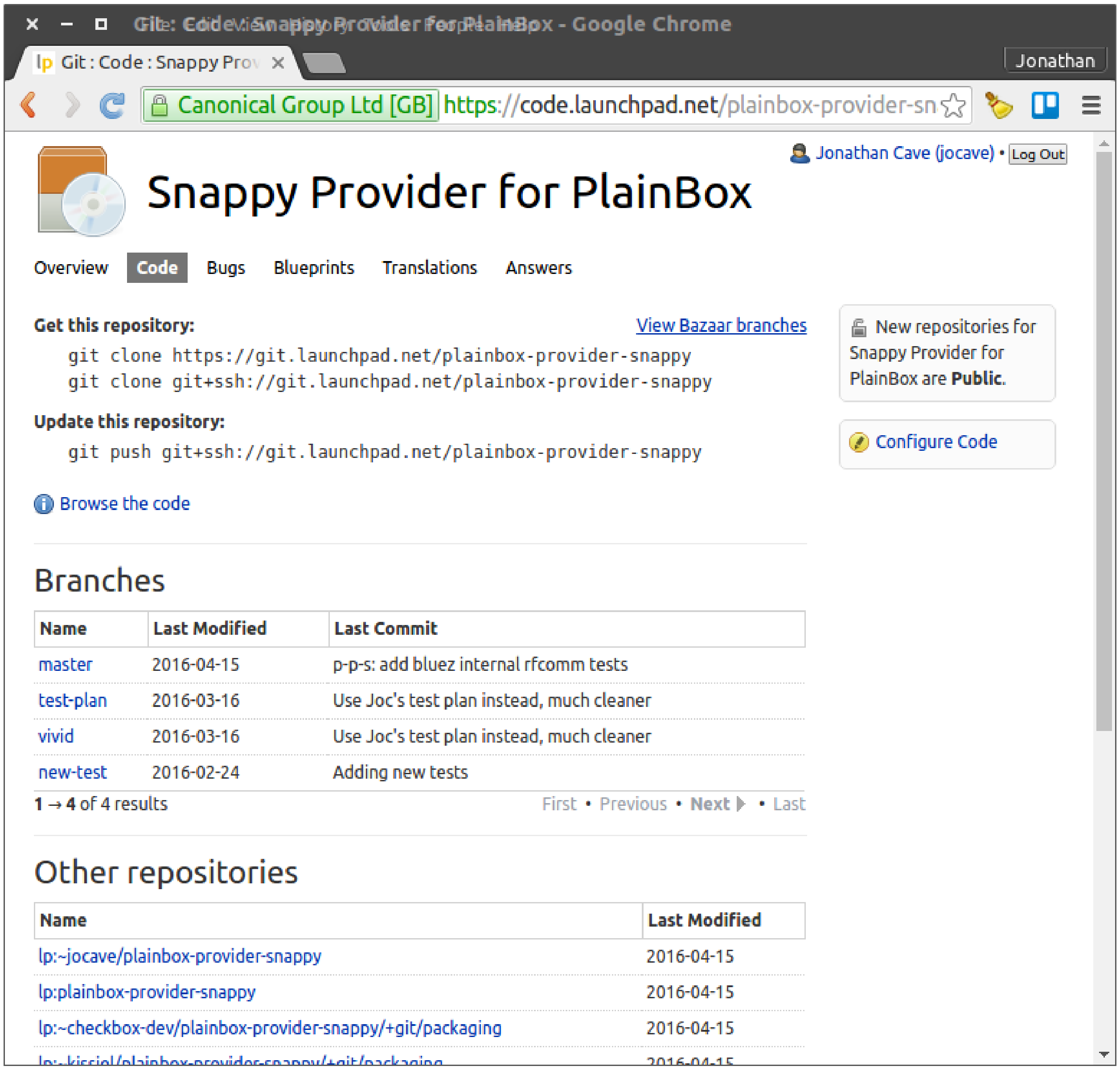
7. Submit your modifications to the project

To push code, report bugs etc. you will require a launchpad account: https://login.launchpad.net/

Once you have an account you will be able to push code up to Launchpad. You can they request a merge in to the master repository. To get the code to Launchpad follow these steps:

```
$ git add <file>
$ git commit -m "Adds a test for..."
$ git remote add my-repo git+ssh://git.launchpad.net/~<USERNAME>/plainbox-provider-
↪snappy
$ git push my-repo <NEW-BRANCH>
```

If you navigate to the plainbox-provider-snappy project on launchpad you should now see your repository listed under the “Other repositories” section. Here you can see my (jocave) personal repository listed at the top:



Snappy Provider for PlainBox

Get this repository:

```
git clone https://git.launchpad.net/plainbox-provider-snappy
git clone git+ssh://git.launchpad.net/plainbox-provider-snappy
```

Update this repository:

```
git push git+ssh://git.launchpad.net/plainbox-provider-snappy
```

New repositories for Snappy Provider for PlainBox are **Public**.

Branches

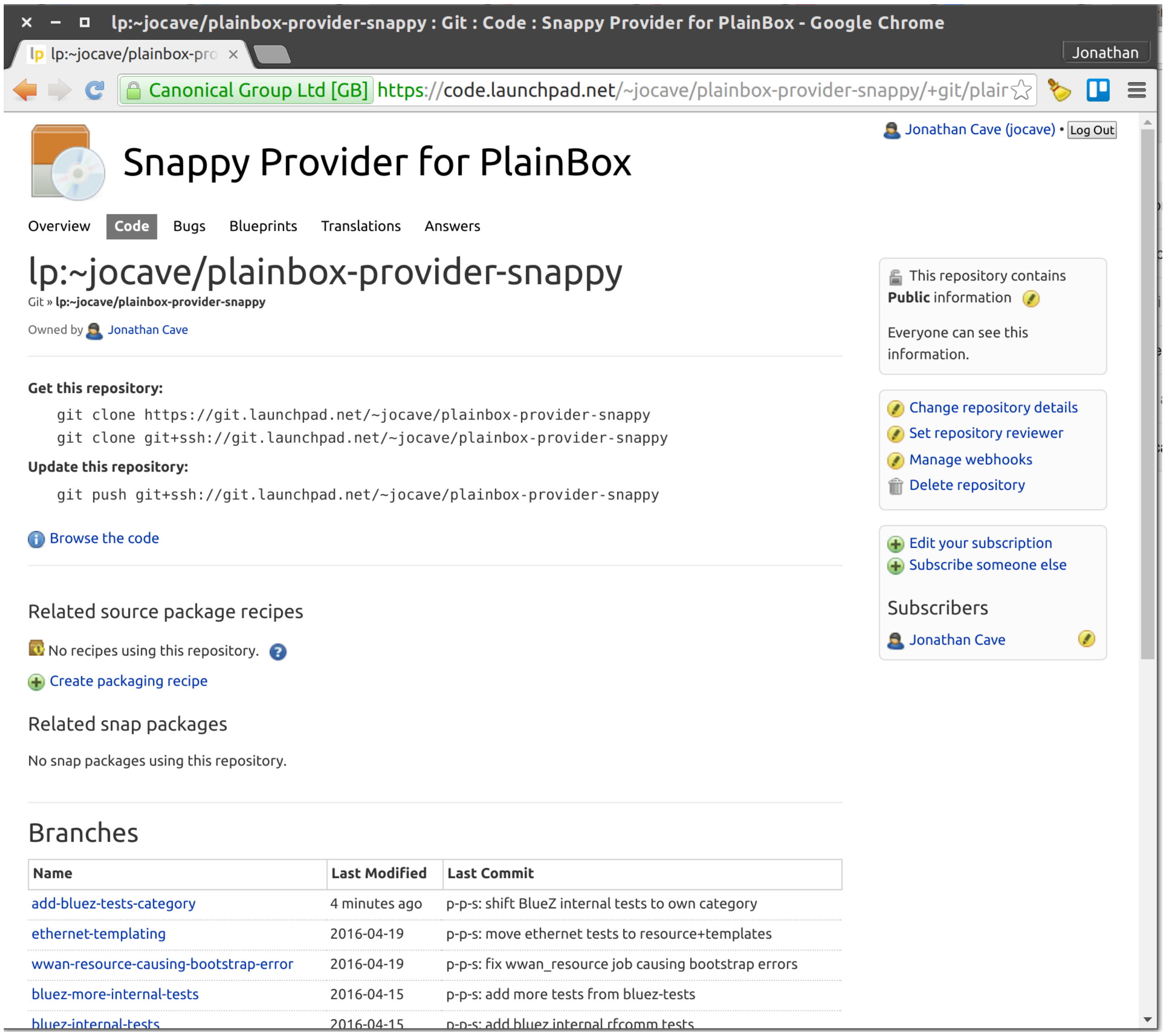
Name	Last Modified	Last Commit
master	2016-04-15	p-p-s: add bluez internal rfcomm tests
test-plan	2016-03-16	Use Joc's test plan instead, much cleaner
vivid	2016-03-16	Use Joc's test plan instead, much cleaner
new-test	2016-02-24	Adding new tests

1 → 4 of 4 results   First • Previous • **Next** • Last

Other repositories

Name	Last Modified
lp:~jocave/plainbox-provider-snappy	2016-04-15
lp:plainbox-provider-snappy	2016-04-15
lp:~checkbox-dev/plainbox-provider-snappy/+git/packaging	2016-04-15
lp:~kissiel/plainbox-provider-snappy/+git/packaging	2016-04-15

Clicking on your repository will take you to an overview page listing all your branches:


lp:~jocave/plainbox-provider-snappy : Git : Code : Snappy Provider for PlainBox - Google Chrome

https://code.launchpad.net/~jocave/plainbox-provider-snappy/+git/plair

Jonathan Cave (jocave) • Log Out

Snappy Provider for PlainBox

Overview Code Bugs Blueprints Translations Answers

lp:~jocave/plainbox-provider-snappy

Git » lp:~jocave/plainbox-provider-snappy

Owned by Jonathan Cave

Get this repository:

```
git clone https://git.launchpad.net/~jocave/plainbox-provider-snappy
git clone git+ssh://git.launchpad.net/~jocave/plainbox-provider-snappy
```

Update this repository:

```
git push git+ssh://git.launchpad.net/~jocave/plainbox-provider-snappy
```

Browse the code

Related source package recipes

No recipes using this repository.

Create packaging recipe

Related snap packages

No snap packages using this repository.

Branches

Name	Last Modified	Last Commit
add-bluez-tests-category	4 minutes ago	p-p-s: shift BlueZ internal tests to own category
ethernet-templating	2016-04-19	p-p-s: move ethernet tests to resource+templates
wwan-resource-causing-bootstrap-error	2016-04-19	p-p-s: fix wwan_resource job causing bootstrap errors
bluez-more-internal-tests	2016-04-15	p-p-s: add more tests from bluez-tests
bluez-internal-tests	2016-04-15	p-p-s: add bluez internal rfcomm tests

This repository contains **Public** information

Everyone can see this information.

Change repository details

Set repository reviewer

Manage webhooks

Delete repository

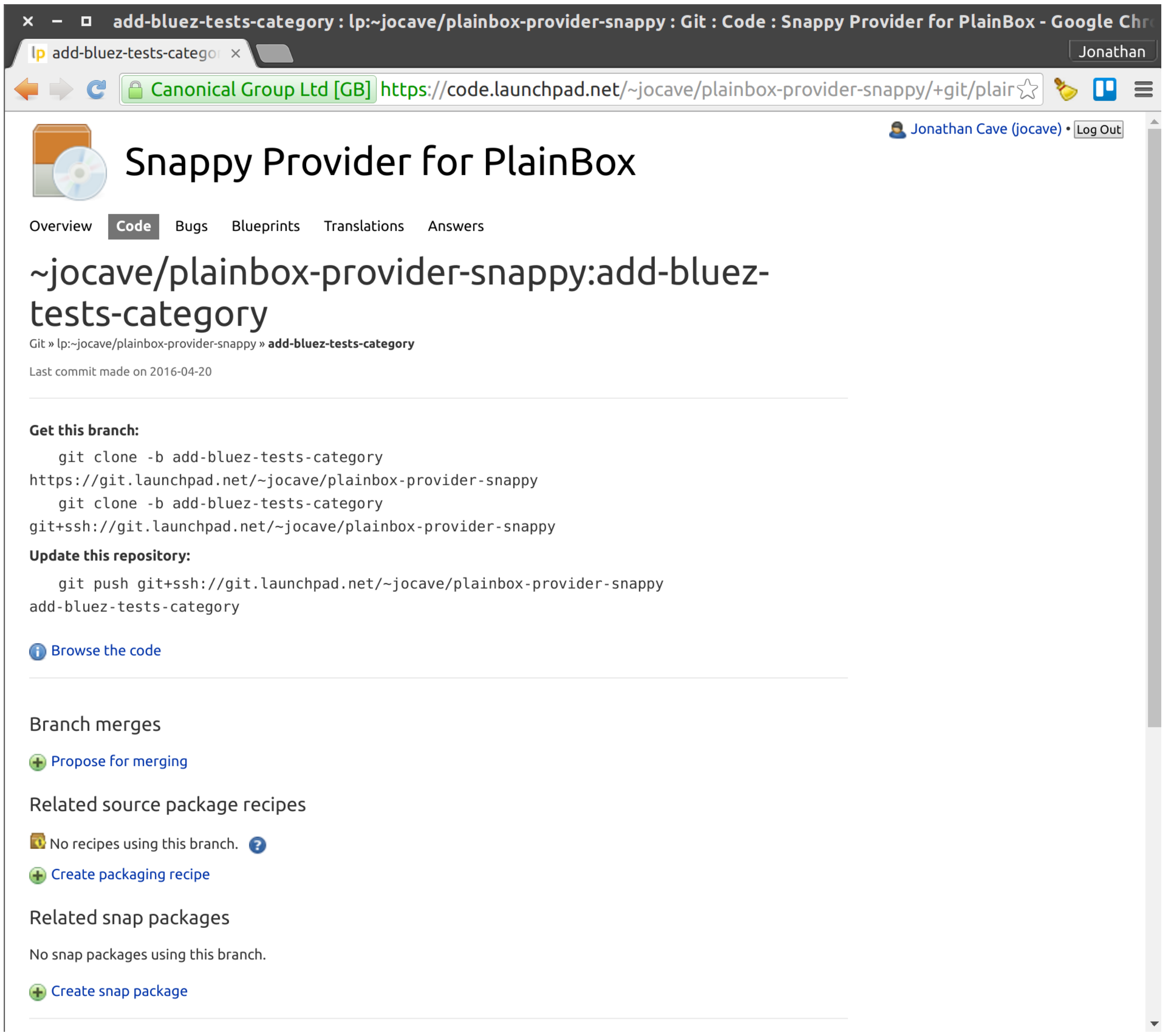
Edit your subscription

Subscribe someone else

Subscribers

Jonathan Cave

Click on the branch you have uploaded and there will be an option to “Propose for merging”.


add-bluez-tests-category : lp:~jocave/plainbox-provider-snappy : Git : Code : Snappy Provider for PlainBox - Google Chr
add-bluez-tests-catego
Jonathan
Canonical Group Ltd [GB] https://code.launchpad.net/~jocave/plainbox-provider-snappy/+git/plair
Jonathan Cave (jocave) • Log Out
Snappy Provider for PlainBox
Overview
Code
Bugs
Blueprints
Translations
Answers

~jocave/plainbox-provider-snappy:add-bluez-tests-category

Git » lp:~jocave/plainbox-provider-snappy » **add-bluez-tests-category**

Last commit made on 2016-04-20

Get this branch:

```
   git clone -b add-bluez-tests-category
https://git.launchpad.net/~jocave/plainbox-provider-snappy
   git clone -b add-bluez-tests-category
git+ssh://git.launchpad.net/~jocave/plainbox-provider-snappy
```

Update this repository:

```
   git push git+ssh://git.launchpad.net/~jocave/plainbox-provider-snappy
add-bluez-tests-category
```

Browse the code

Branch merges

Propose for merging

Related source package recipes

No recipes using this branch.

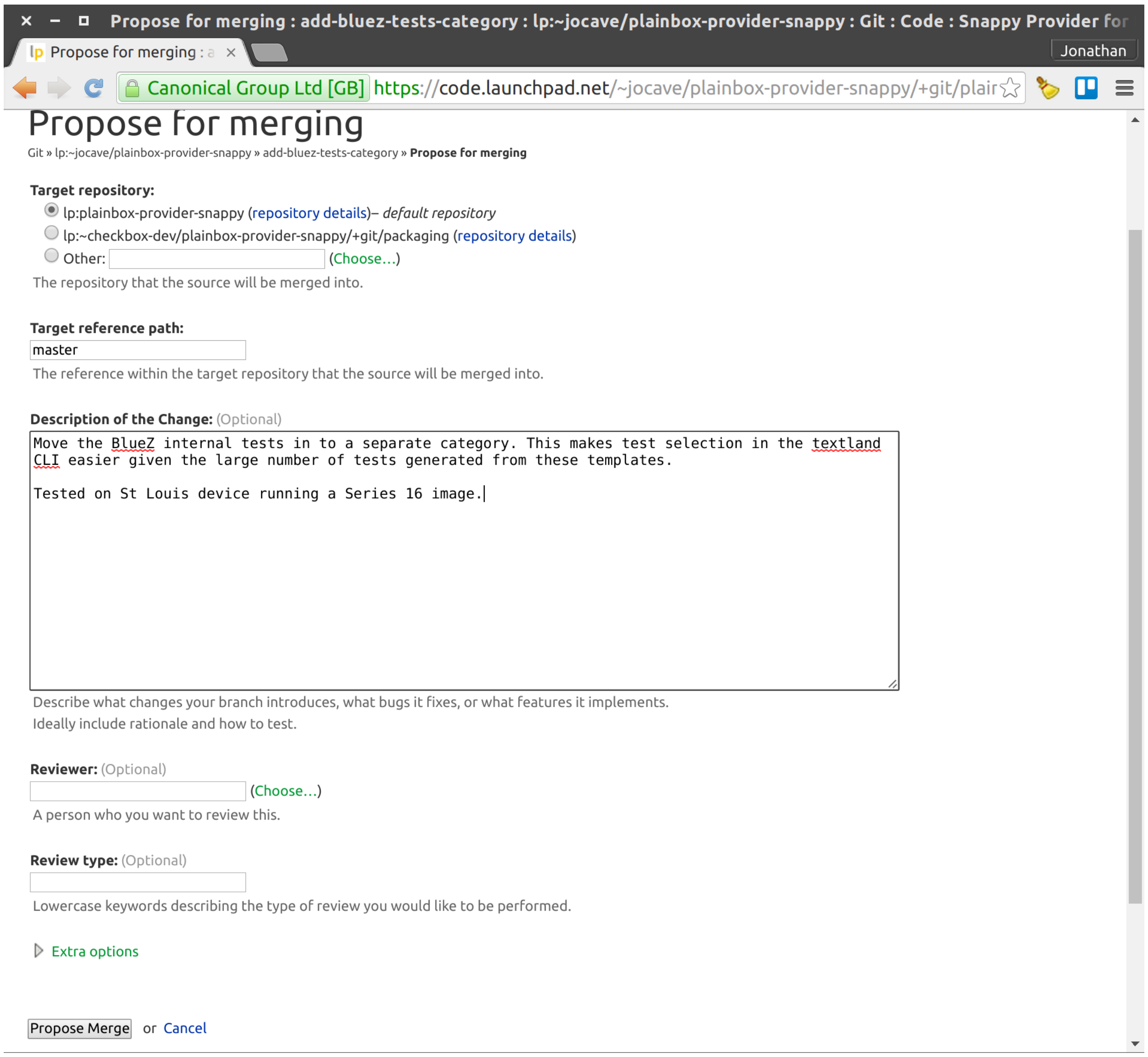
Create packaging recipe

Related snap packages

No snap packages using this branch.

Create snap package

Select this and fill out the form as follows:


Propose for merging : add-bluez-tests-category : lp:~jocave/plainbox-provider-snappy : Git : Code : Snappy Provider for
Jonathan
Canonical Group Ltd [GB] https://code.launchpad.net/~jocave/plainbox-provider-snappy/+git/plair

Propose for merging

Git » lp:~jocave/plainbox-provider-snappy » add-bluez-tests-category » **Propose for merging**

Target repository:

● lp:plainbox-provider-snappy (repository details)– *default repository*
○ lp:~checkbox-dev/plainbox-provider-snappy/+git/packaging (repository details)
○ Other: (Choose…)

The repository that the source will be merged into.

Target reference path:

master

The reference within the target repository that the source will be merged into.

Description of the Change: (Optional)

```
Move the BlueZ internal tests in to a separate category. This makes test selection in the textland
CLI easier given the large number of tests generated from these templates.

Tested on St Louis device running a Series 16 image.
```

Describe what changes your branch introduces, what bugs it fixes, or what features it implements.
Ideally include rationale and how to test.

Reviewer: (Optional)

(Choose…)

A person who you want to review this.

Review type: (Optional)

Lowercase keywords describing the type of review you would like to be performed.

Extra options

Propose Merge or Cancel

Members of the team that maintain the project will be alerted to the Merge Request and will review it for landing.

Running Checkbox on Ubuntu Core

Introduction

Checkbox is a hardware testing tool developed by Canonical for certifying hardware with Ubuntu. Checkbox is free software and is available at http://launchpad.net/checkbox.

To support the release of devices running snappy Ubuntu Core, Canonical has produced versions of Checkbox tailored specifically for these systems.

This document aims to provide the reader with enough information to install and run Checkbox on an Ubuntu Core system, and how to view/interpret/submit test results.

Installation

Installing Ubuntu Core on KVM

Download the following release of Ubuntu Core (or the one provided by Canonical for your project):

```
$ wget http://people.canonical.com/~mvo/all-snaps/16/all-snaps-pc.img.xz
```

Install it on a snappy DUT or boot the img file in kvm with:

```
$ unxz all-snaps-pc.img.xz
$ kvm -m 4096 -redir tcp:8022::22 ./all-snaps-pc.img
```

Log in as the ubuntu user (password ubuntu):

```
$ ssh -p 8022 ubuntu@localhost
```

Perform a snappy update:

```
$ sudo snap refresh
```

Installing Checkbox Snap

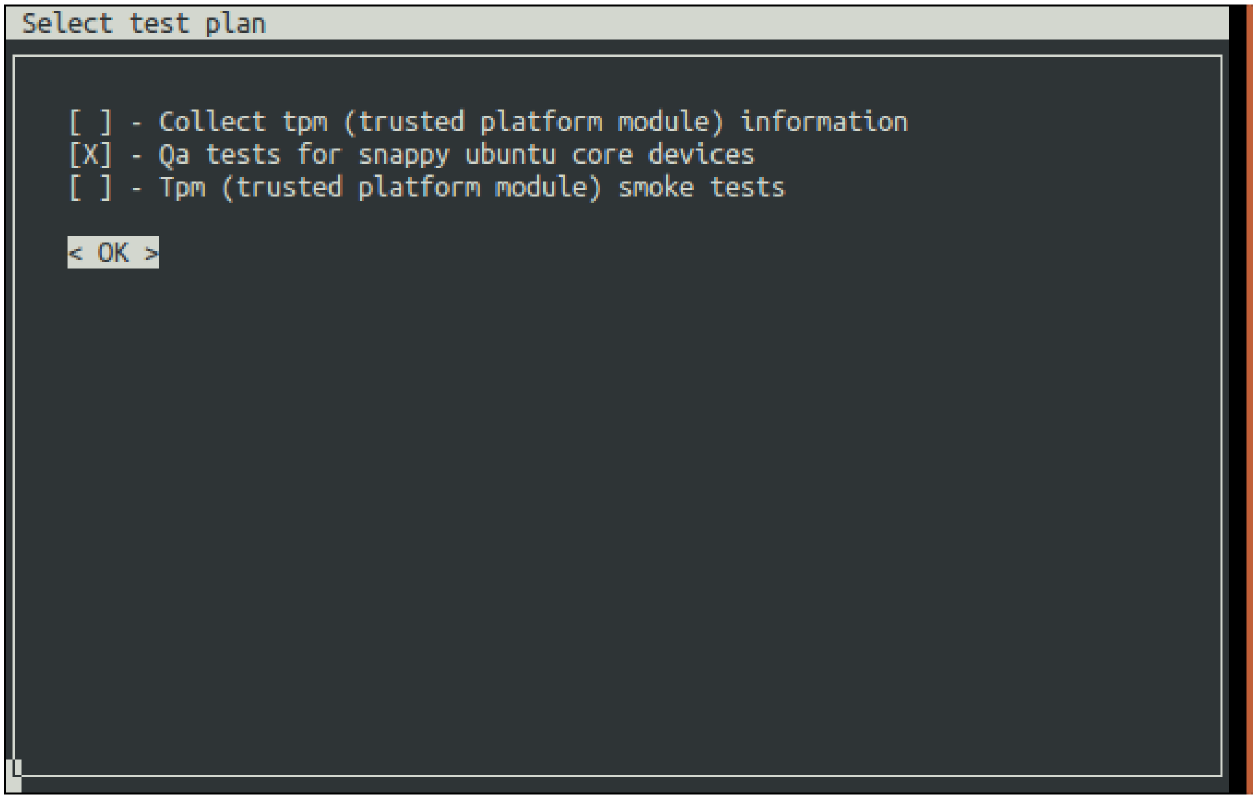
Now you are ready to install the checkbox snap, install it straight from the store.

> $ sudo snap install checkbox-snappy –edge –devmode

Running Checkbox

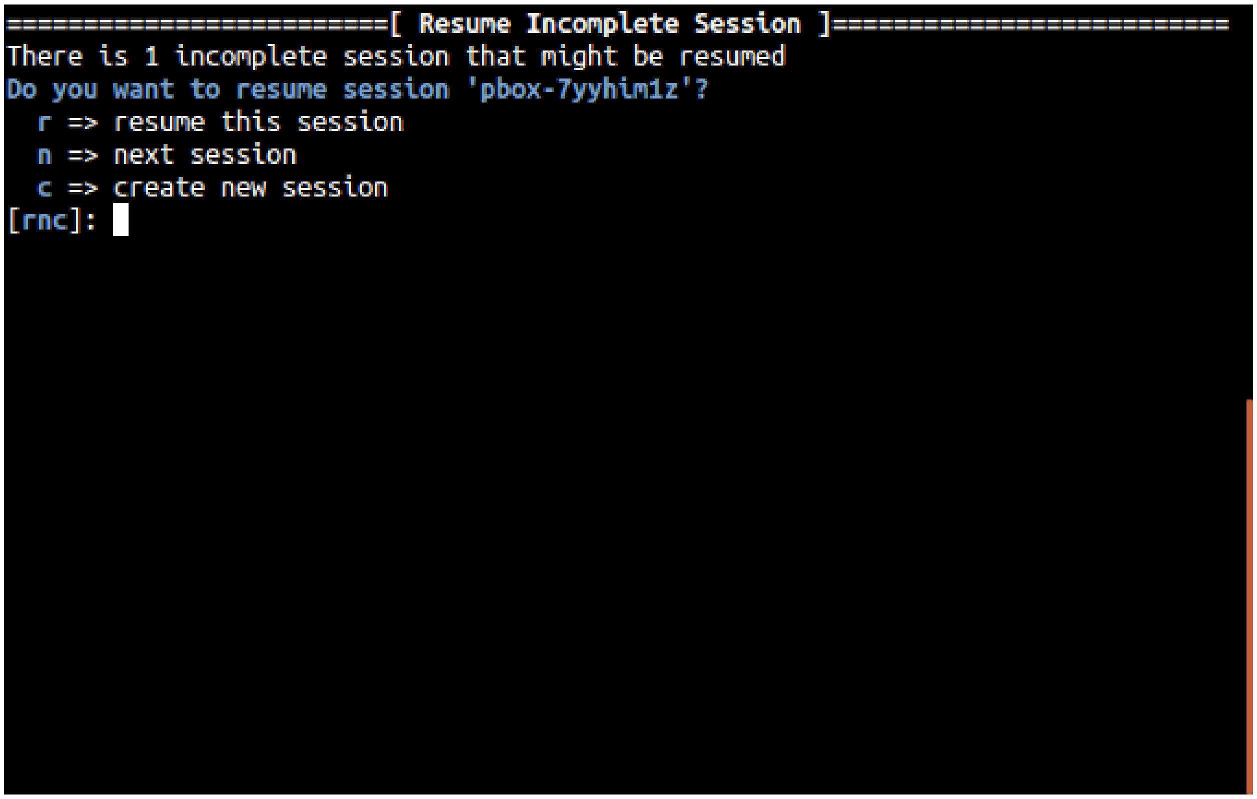
Simply launch checkbox using:

```
$ checkbox-snappy.test-runner
```



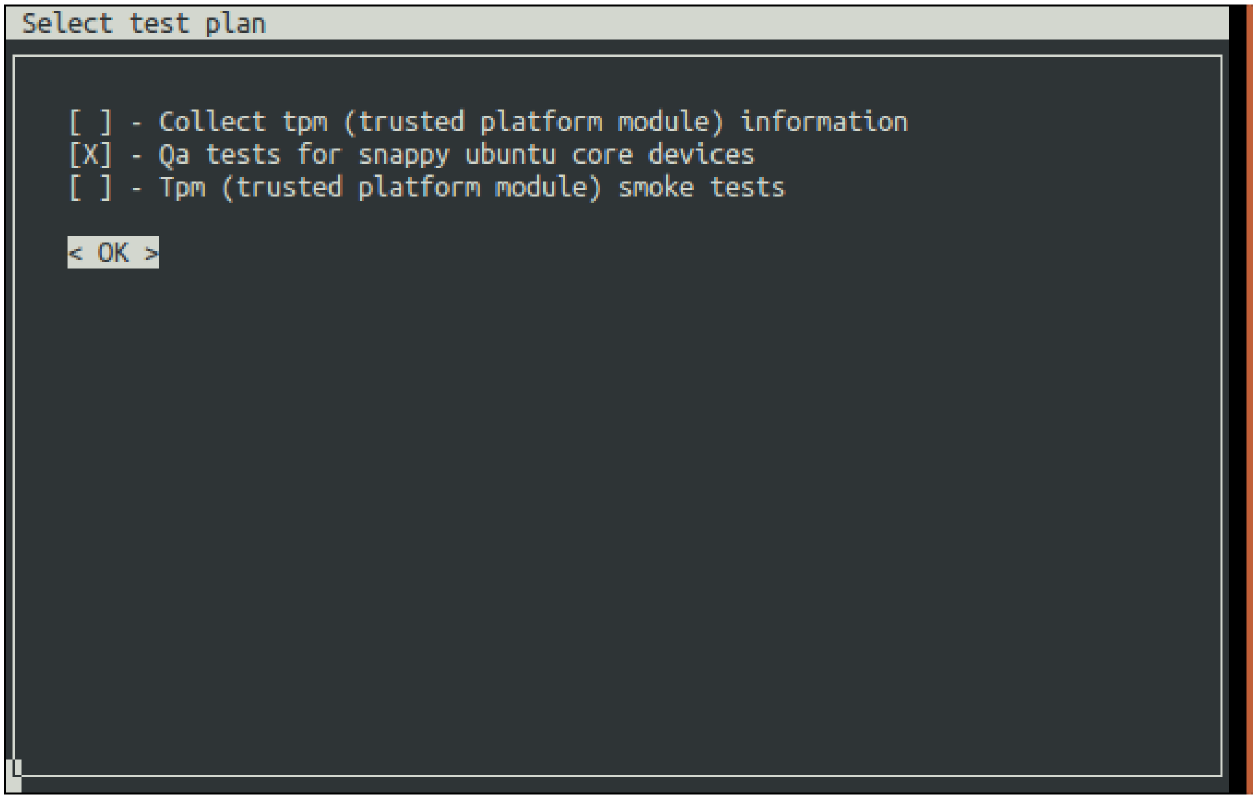
```
Select test plan

  [ ] - Collect tpm (trusted platform module) information
  [X] - Qa tests for snappy ubuntu core devices
  [ ] - Tpm (trusted platform module) smoke tests

  < OK >
```

Checkbox keeps track are previous test runs, if a session is not completed, you'll be asked to resume your previous run or create a new session:



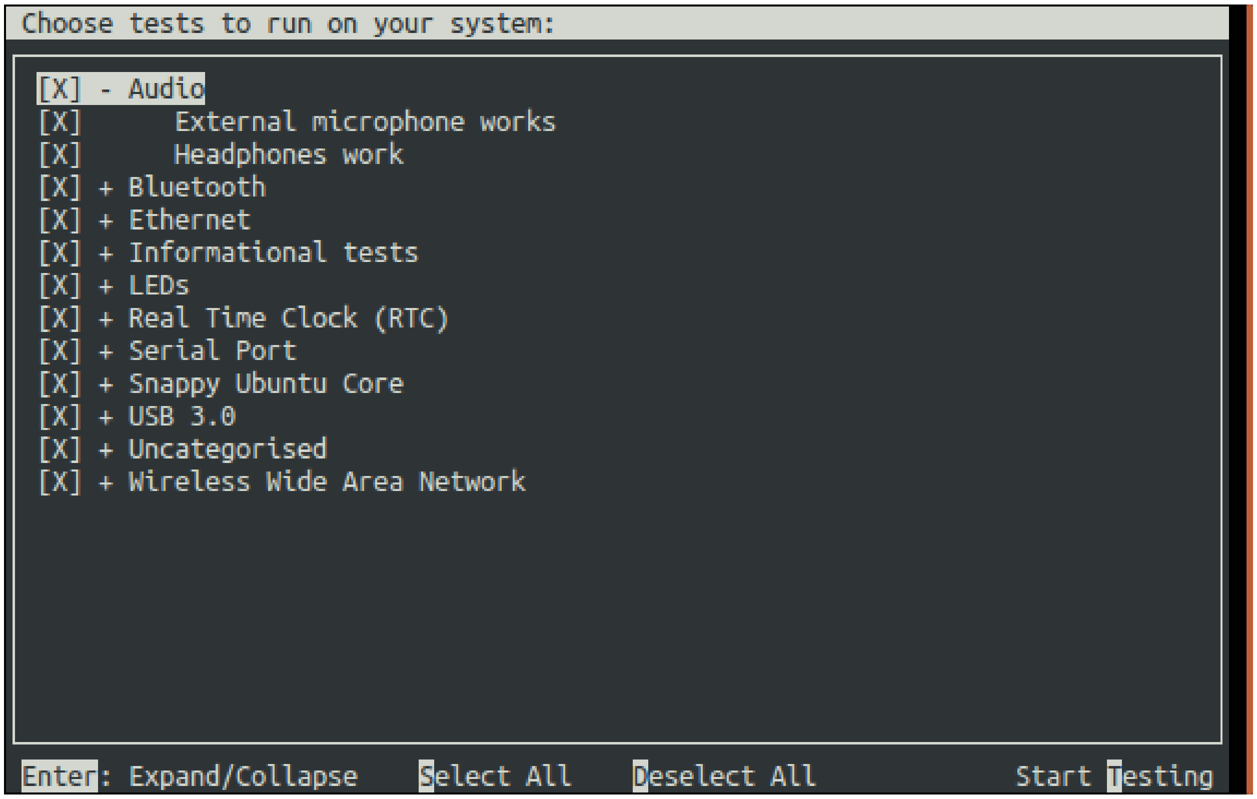
```
========================[ Resume Incomplete Session ]=========================
There is 1 incomplete session that might be resumed
Do you want to resume session 'pbox-7yyhim1z'?
  r => resume this session
  n => next session
  c => create new session
[rnc]: 
```

The first selection screen will ask you to select a test plan to run:



```
Select test plan

  [ ] - Collect tpm (trusted platform module) information
  [X] - Qa tests for snappy ubuntu core devices
  [ ] - Tpm (trusted platform module) smoke tests

  < OK >
```

Move the selection with the arrow keys, select with space and confirm your choice by moving the selection to <OK> and press Enter. The next screen will allow you to fine tune the tests you want to run:



```
Choose tests to run on your system:

 [X] - Audio
 [X]      External microphone works
 [X]      Headphones work
 [X] + Bluetooth
 [X] + Ethernet
 [X] + Informational tests
 [X] + LEDs
 [X] + Real Time Clock (RTC)
 [X] + Serial Port
 [X] + Snappy Ubuntu Core
 [X] + USB 3.0
 [X] + Uncategorised
 [X] + Wireless Wide Area Network

Enter: Expand/Collapse    Select All    Deselect All            Start Testing
```

Tests are grouped by categories, Expand/Collapse with Enter, select/unselect with space (also works on categories). Press S to select all and D to Deselect all the tests.

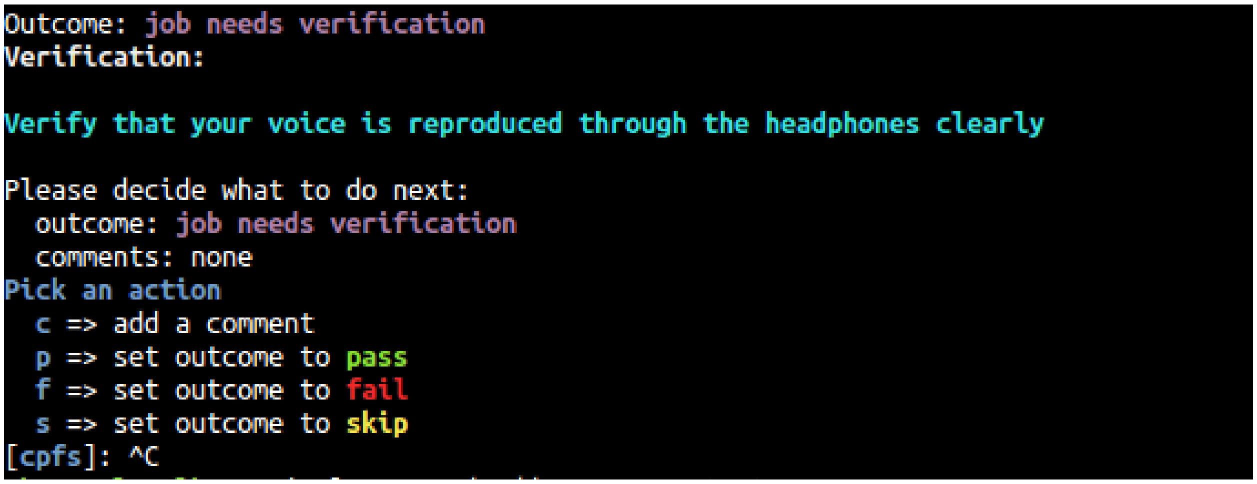
Start the tests by pressing T.

Checkbox is a test runner able to process fully automated tests/commands and tests requiring user interaction (whether to setup or plug something to the device, e.g USB insertion or to confirm that the device acts as expected, e.g a led blinks).

Please refer to the checkbox documentation to learn more about the supported type of tests.

A fully automated test will stream stdout/stderr to your terminal allowing you to immediately look at the i/o logs (if the session is run interactively). Attachments jobs are treated differently as they could generate lots of i/o. Therefore their outputs are hidden by default.

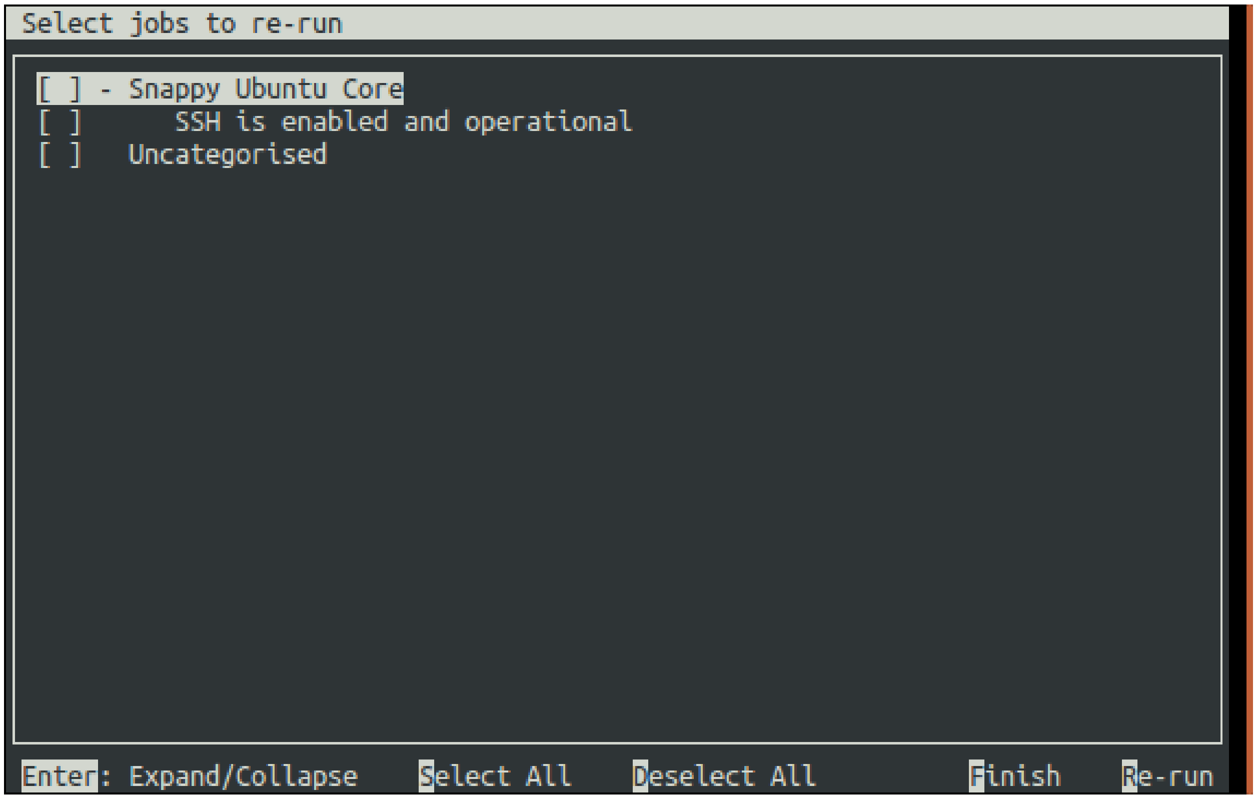
Interactive jobs will pause the test runner and details the steps to complete the test:



```
Outcome: job needs verification
Verification:

Verify that your voice is reproduced through the headphones clearly

Please decide what to do next:
  outcome: job needs verification
  comments: none
Pick an action
  c => add a comment
  p => set outcome to pass
  f => set outcome to fail
  s => set outcome to skip
[cpfs]: ^C
```

Getting Results

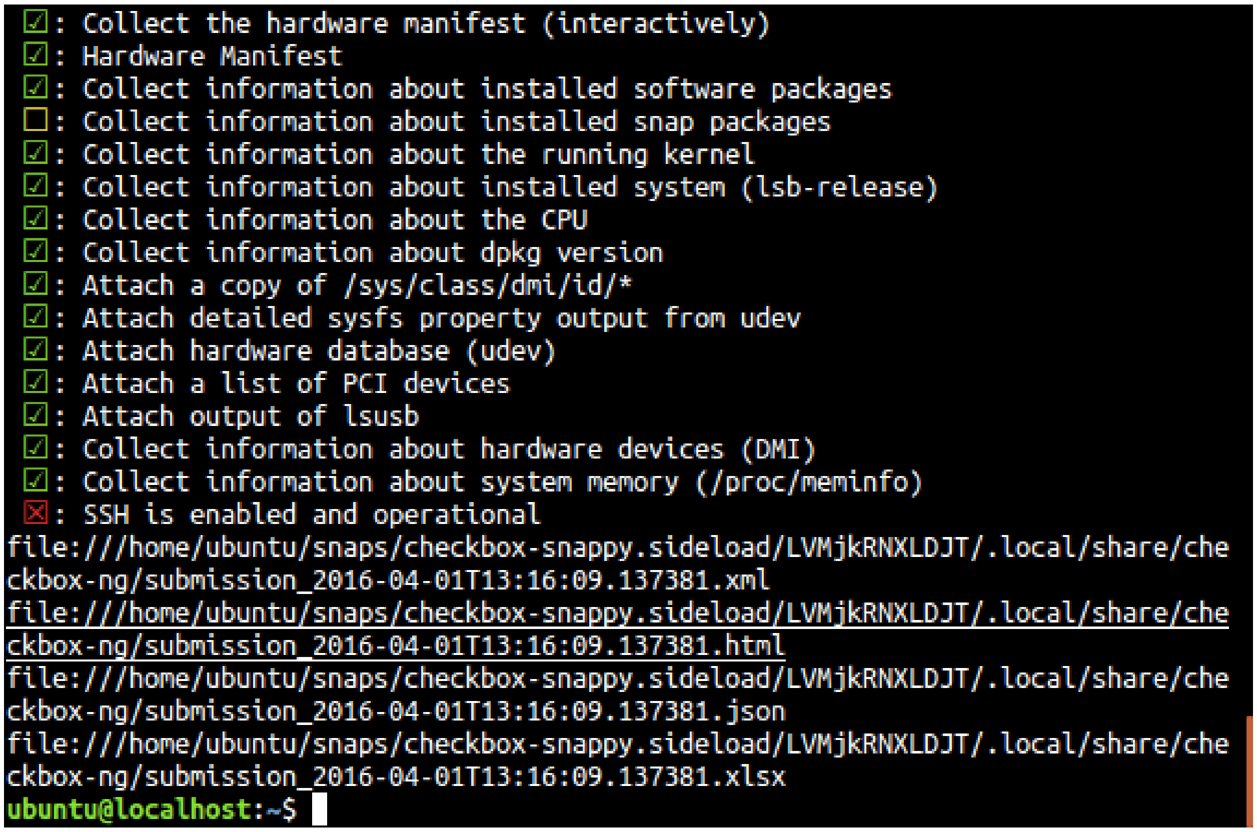
When the test selection has been run, the first displayed screen will allow you to re-run failed jobs (e.g. the wireless access point was not properly configured):



```
Select jobs to re-run

[ ] - Snappy Ubuntu Core
[ ]       SSH is enabled and operational
[ ]   Uncategorised

Enter: Expand/Collapse    Select All    Deselect All          Finish    Re-run
```

Commands to select the tests to rerun are the same used to select tests in the first selection screen. Here you can rerun your selection with R or finish the session by pressing F.

Checkbox will then print the the test results in the terminal and save them in different formats locally on the device (and print their respective filenames):



```
☑: Collect the hardware manifest (interactively)
☑: Hardware Manifest
☑: Collect information about installed software packages
☐: Collect information about installed snap packages
☑: Collect information about the running kernel
☑: Collect information about installed system (lsb-release)
☑: Collect information about the CPU
☑: Collect information about dpkg version
☑: Attach a copy of /sys/class/dmi/id/*
☑: Attach detailed sysfs property output from udev
☑: Attach hardware database (udev)
☑: Attach a list of PCI devices
☑: Attach output of lsusb
☑: Collect information about hardware devices (DMI)
☑: Collect information about system memory (/proc/meminfo)
☒: SSH is enabled and operational
file:///home/ubuntu/snaps/checkbox-snappy.sideload/LVMjkRNXLDJT/.local/share/che
ckbox-ng/submission_2016-04-01T13:16:09.137381.xml
file:///home/ubuntu/snaps/checkbox-snappy.sideload/LVMjkRNXLDJT/.local/share/che
ckbox-ng/submission_2016-04-01T13:16:09.137381.html
file:///home/ubuntu/snaps/checkbox-snappy.sideload/LVMjkRNXLDJT/.local/share/che
ckbox-ng/submission_2016-04-01T13:16:09.137381.json
file:///home/ubuntu/snaps/checkbox-snappy.sideload/LVMjkRNXLDJT/.local/share/che
ckbox-ng/submission_2016-04-01T13:16:09.137381.xlsx
ubuntu@localhost:~$
```

The resulting reports can be easily pulled from the system via SCP, or by simply copying to a USB stick.

Creating a custom Checkbox application for Ubuntu Core testing

This guide describes how to create a custom Checkbox application for testing a new project (project meaning a new system that we want to test with Checkbox).

Initialize the project

Creating your working directory and initializing the projects. Make sure you have at least snapcraft version 2.13 (available in Ubuntu 16.04 or newer).:

```
mkdir checkbox-myproject
cd checkbox-myproject
snapcraft init
git init
```

You will now have a `snapcraft.yaml` file in the `snap` directory. Modify it and insert your title, description, version.

Listing 3.1: snap/snapcraft.yaml

```
name: checkbox-myproject
version: 1
summary: Checkbox tool for MyProject
description: Checkbox tool for MyProject
grade: devel
confinement: strict
```

Adding parts

Add the basic reusable snappy provider parts.

Listing 3.2: snap/snapcraft.yaml

```
(...)
parts:
    plainbox-provider-snappy:
        after: [plainbox-provider-snappy-resource]
    plainbox-provider-snappy-resource:
        after: [plainbox-dev, checkbox-support-dev, checkbox-ng-dev]

    network-tools:
        plugin: nil
        stage-packages:
            - network-manager
            - modemmanager
            - hostapd
            - iw
        snap:
            - usr/bin/mmcli
            - usr/lib/*/libmm-glib.so*
            - usr/bin/nmcli
            - usr/lib/*/libnm*
            - usr/sbin/hostapd
            - sbin/iw
```

Create a device/project specific provider

```
$ plainbox startprovider --empty com.canonical.qa.myproject:
plainbox-provider-myproject
```

The directory name for the provider is quite a mouthful, let’s change it to something more manageable.

```
$ mv com.canonical.qa.myproject:plainbox-provider-myproject
plainbox-provider-myproject
```

This new provider has to also be included as a part of the snap

```
:caption: snap/snapcraft.yaml
:name: snapcraft.yaml-with-custom-provider

(...)
parts:
    plainbox-provider-myproject:
        plugin: plainbox-provider
        source: ./plainbox-provider-myproject
        after: [plainbox-provider-snappy]
```

Create your new test plans (and jobs to go in them)

Edit the plainbox-provider-myproject provider by adding jobs and particularly test plans that list all the jobs that you want to run.

By convention units reside in .pxu files in the `units` directory of the provider. Let's create one

```
$ cd plainbox-provider-myproject
$ mkdir units
```

Let's add a job from *Checkbox tutorials*

Listing 3.3: units/jobs.pxu

```
id: my-first-job
_summary: 10GB available in $HOME
_description:
    this test checks if there's at least 10gb of free space in user's home
        directory
plugin: shell
estimated_duration: 0.01
command: [ `df -B 1G --output=avail $HOME |tail -n1` -gt 10 ]
```

You may read more on how to write jobs here: *Job Unit*

It is a good practice to group jobs in test plans, here's one that will include the `my-first-job`

Listing 3.4: unit/test-plan.pxu

```
unit: test plan
id: my-project-custom
_name: MyProject tests
_description:
    This test plan includes all test related to MyProject
include:
    my-first-job
```

You may read more on test plans here: *Test Plan Unit*

Reusing existing provider(s)

It's best not to duplicate stuff, so if the test you want to run already exists in another provider it is best to include that provider in the snap, and include the test, or whole test plans from that provider in your new testing project.

Let's reuse disk tests from the "plainbox-provider-snappy" provider that we already have as a part of the snap. All we need is a test plan that will include both reused disk tests and the new custom ones.

Listing 3.5: unit/test-plan.pxu

```
id: my-project-all-tests
_name: All MyProject tests
```

```
_description:
    This test plan includes some disk tests from plainbox-provider-snappy
    and the my-first-job test.
include:
    com.canonical.certification::disk/detect
    com.canonical.certification::disk/stats_.*
    my-first-job
```

You can also include the whole *external* test plan. Let's reuse the CPU testing suite from plainbox-provider-snappy.

Listing 3.6: unit/test-plan.pxu

```
unit: test plan
id: my-project-all-tests
_name: All MyProject tests
_description:
    This test plan includes some disk tests from plainbox-provider-snappy
    and the my-first-job test.
include:
    com.canonical.certification::disk/detect
    com.canonical.certification::disk/stats_.*
    my-first-job
nested_part:
    com.canonical.certification::cpu-full
```

Create Checkbox Launchers configurations

Launchers help to predefine how Checkbox should run. Read more here: *Checkbox launchers tutorial*

First, let's leave the provider directory and go back to the `checkbox-myproject`.

```
$ cd ..
```

and write the first launcher

Listing 3.7: launchers/myproject-test-runner

```
#!/usr/bin/env checkbox-cli-wrapper
[launcher]
app_id = com.canonical.qa.myproject:checkbox
launcher_version = 1
stock_reports = text, submission_files

[test plan]
filter = *myproject*, *tpm-smoke-tests
```

Create wrapper scripts

We currently need wrapper scripts to discover providers, set up the execution environment and work around a few other snappy issues. Add one like this:

Listing 3.8: launchers/checkbox-cli-wrapper:

```
#!/bin/bash
```

```
export PATH="$PATH:$SNAP/usr/sbin"
exec python3 $(which checkbox-cli) "$@"
```

Now we need to make the launchers executable

```
chmod +x launchers/*
```

Listing 3.9: snap/snapcraft.yaml

```
(...)
launchers:
    plugin: dump
    source: launchers/
    organize:
        '*': bin/
```

Declare the launchers to be Apps that exist in your Snap

Listing 3.10: snap/snapcraft.yaml

```
(...)
apps:
    myproject-test-runner:
        command: bin/myproject-test-runner
```

What's left is to snap it all together!

```
$ snapcraft
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

R

RFC
    RFC 822, 36