SQUAD Documentation

Release 1.89

Linaro

Jul 25, 2024

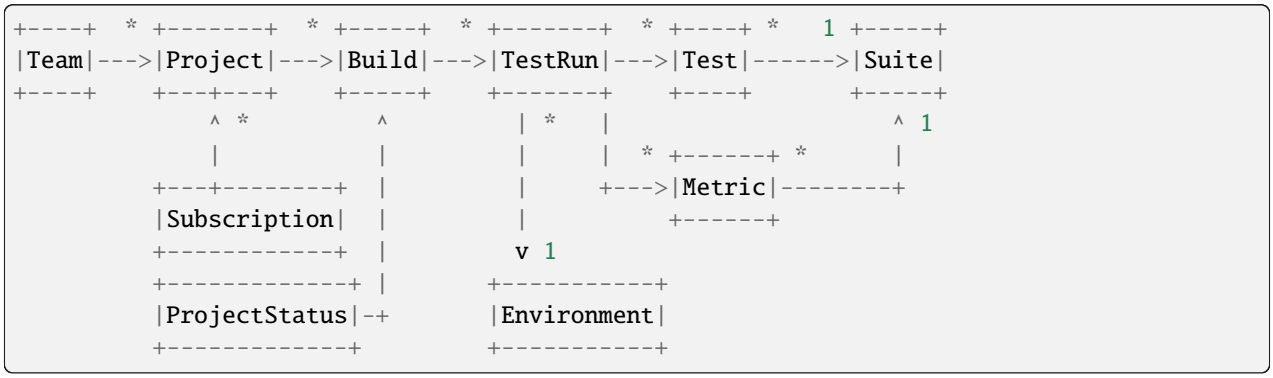
CONTENTS:

CHAPTER

ONE

INTRODUCTION: DATA MODEL AND USAGE

1.1 Core data model



SQUAD is multi-group and multi-project. Each group can have multiple projects. For each project, you can have multiple builds, and for each build, multiple test runs. Each test run can include multiple test results, which can be either pass/fail results, or metrics, containing one or more measurement values. Test and metric results can belong to a Suite, which is a basically used to group and analyze results together. Every test suite must be associated with exactly one Environment, which describes the environment in which the tests were executed, such as hardware platform, hardware configuration, OS, build settings (e.g. regular compilers vcs optimized compilers), etc. Results are always organized by environments, so we can compare apples to apples.

Projects can have subscriptions, which are either users or manually-entered email addreses that should be notified about important events such as changing test results. ProjectStatus records the most recent build of a project, against which future results should be compared in search for important events to notify subscribers about. SQUAD also supports a metric threshold system, which will send notification to project subscribers if the test result metrics exceed a certain value. The threshold values will also appear in the charts. Projects have the *project_settings* field for any specific configuration it might require.

Builds can be compared against baselines exposing regressions and fixes. Visit `/_/comparebuilds/`, select a project then two builds from it. This comparison will go over each build's tests and find all different states from each. For instance if a test failed in baseline but passes in a more current one, this is considered to be a `"fix"`. The opposite is called `"regression"`. The concept can vary for other transitions.

1.2 Submitting results

The API is the following

POST /api/submit/:group/:project/:build/:environment

- `:group` is the group identifier. It must exist previously.
- `:project` is the project identifier. It must exist previously.
- `:build` is the build identifier. It can be a git commit hash, a Android manifest hash, or anything really. Extra information on the build can be submitted as an attachment. If a build timestamp is not informed there, the time of submission is assumed.
- `:environment` is the environmenr identitifer. It will be created automatically if does not exist before.

All of the above identifiers (`:group`, `:project`, `:build`, and `:environment`) must match the regular expression `[a-zA-Z0-9][a-zA-Z0-9_.-]*`.

The test data files must be submitted as either file attachments, or as regular `POST` parameters. . The following files are supported:

- `tests`: test results data
- `metrics`: metrics data
- `metadata`: metadata about the test run
- `attachment`: arbitrary file attachments. Multiple attachments can be submitted by providing this parameter multiple times.

See *Input file formats* below for details on the format of the data files.

Example with test data as file uploads:

```
$ curl \
    --header "Authorization: token $SQUAD_TOKEN" \
    --form tests=@/path/to/test-results.json \
    --form metrics=@/path/to/metrics.json \
    --form metadata=@/path/to/metadata.json \
    --form log=@/path/to/log.txt \
    --form attachment=@/path/to/screenshot.png \
    --form attachment=@/path/to/extra-info.txt \
    https://squad.example.com/api/submit/my-group/my-project/x.y.z/my-ci-env
```

Example with test data as regular `POST` parameters:

```
$ curl \
    --header "Authorization: token $SQUAD_TOKEN" \
    --form tests='{"test1": "pass", "test2": "fail"}' \
    --form metrics='{"metric1": 21, "metric2": 4}' \
    --form metadata='{"foo": "bar", "baz": "qux", "job_id": 123}' \
    --form log='log text ...' \
    --form attachment=@/path/to/screenshot.png \
    --form attachment=@/path/to/extra-info.txt \
    https://squad.example.com/api/submit/my-group/my-project/x.y.z/my-ci-env
```

Example with test data using Python's requests library:

```
import json
import requests
import os

tests = json.dumps({"test1": "pass", "test2": "fail"})
metrics = json.dumps({"metric1": 21, "metric2": 4})
metadata = json.dumps({"foo": "bar", "baz": "qux", "job_id": 123})
log = 'log text ...'

headers = {"Authorization": f"token {os.getenv('SQUAD_TOKEN')}"}
url = 'https://squad.example.com/api/submit/my-group/my-project/x.y.z/my-ci-env'
data = {"metadata": metadata, "log": log, "tests": tests_file}

result = requests.post(url, headers=headers, data=data)
if not result.ok:
    print(f"Error submitting to qa-reports: {result.reason}: {result.text}")
```

Since test results should always come from automation systems, the API is the only way to submit results into the system. Even manual testing should be automated with a driver program that asks for user input, and them at the end prepares all the data in a consistent way, and submits it to dashboard.

If input data is valid and nothing goes wrong with the request, SQUAD will return 201 as status code and the test run id in the response body.

1.3 Input file formats

1.3.1 Test results

Test results must be posted as JSON, encoded in UTF-8. The JSON data must be a hash (an object, strictly speaking). Test names go in the keys, and values must be either `"pass"` or `"fail"`. Case does not matter, so `"PASS"`/`"FAIL"` will work just fine. Any value that when downcased is not either `"pass"` or `"fail"` will be mapped to `None`/`NULL` and displayed in the UI as *skip*.

Tests that have `"fail"` as results and are known to have any issues are displayed as *xfail* (eXpected-fail).

Tests can be grouped in test suites. For that, the test name must be prefixed with the suite name and a slash (`/`). Therefore, slashes are reserved characters in this context, and cannot be used in test names. There is one exception to this rule. If test name contains square brackets (`[`, `]`) they are considered as test variant. The string inside brackets can contain slashes. Suite names can have embedded slashes in them; so "foo/bar" means suite "foo", test "bar"; and "foo/bar/baz" means suite "foo/bar", test "baz".

Example:

```
{
  "test1": "pass",
  "test2": "pass",
  "testsuite1/test1": "pass",
  "testsuite1/test2": "fail",
  "testsuite2/subgroup1/testA": "pass",
  "testsuite2/subgroup2/testA": "pass",
  "testsuite2/subgroup2/testA[variant/one]": "pass",
  "testsuite2/subgroup2/testA[variant/two]": "pass"
}
```

There is an alternative format for sending results. Since SQUAD supports storing test log in the Test object, passed JSON file can look as follows:

```
{
  "test1": {"result": "pass", "log": "test 1 log"},
  "test2": {"result": "pass", "log": "test 2 log"},
  "testsuite1/test1": {"result": "pass", "log": "test 1 log"},
  "testsuite1/test2": {"result": "fail", "log": "test 2 log"}
}
```

Both forms are supported. In case log entry is missing or simple JSON format is used, logs for each Test object are empty. They can be filled in using plugins.

1.3.2 Metrics

Metrics must be posted as JSON, encoded in UTF-8. The JSON data must be a hash (an object, strictly speaking). Metric names go in the keys, and values must be either a single number, or an array of numbers. In the case of an array of numbers, then their mean will be used as the metric result; the whole set of results will be used where applicable, e.g. to display ranges.

As with test results, metrics can be grouped in suites. For that, the test name must be prefixed with the suite name and a slash (/). Therefore, slashes are reserved characters in this context, and cannot be used in test names. Suite names can have embedded slashes in them; so "foo/bar" means suite "foo", metric "bar"; and "foo/bar/baz" means suite "foo/bar", metric "baz".

Example:

```
{
  "v1": 1,
  "v2": 2.5,
  "group1/v1": [1.2, 2.1, 3.03],
  "group1/subgroup/v1": [1, 2, 3, 2, 3, 1]
}
```

1.3.3 Metadata

Metadata about the test run must be posted in JSON, encoded in UTF-8. The JSON data must be a hash (an object). Keys and values must be strings. The following fields are recognized:

- `build_url`: URL pointing to the origin of the build used in the test run
- `datetime`: timestamp of the test run, as a ISO-8601 date representation, with seconds. This is the representation that `date --iso-8601=seconds` gives you.
- `job_id`: identifier for the test run. Must be unique for the project. **This field is mandatory**
- `job_status`: string identifying the status of the project. SQUAD makes no judgement about its value.
- `job_url`: URL pointing to the original test run.
- `resubmit_url`: URL that can be used to resubmit the test run.
- `suite_versions`: a dictionary with version number strings for suite names used in the tests and metrics data. For example, if you have test suites called "foo" and "bar", their versions can be expressed having metadata that looks like this:

```
{
    # ...
    "suite_versions": {
        "foo": "1.0",
        "bar": "3.1"
    }
}
```

If a metadata JSON file is not submitted, the above fields can be submitted as POST parameters. If a metadata JSON file is submitted, no POST parameters will be considered to be used as metadata.

When sending a proper metadata JSON file, other fields may also be submitted. They will be stored, but will not be handled in any specific way.

1.4 CI loop integration (optional)

SQUAD can integrate with existing automation systems to participate in a Continuous Integration (CI) loop through its CI subsystem. For more details check *CI module in SQUAD*.

1.4.1 Default auth group 'squad'

SQUAD creates by default an auth group with most of the permissions required for authenticated/registered users to view, add, change and delete objects in the projects they have access to. The name of the group is 'squad' by default. All newly created users therefrom are automatically added to this group to eleviate the need for manual intervention to add a user each time one is created.

CHAPTER

TWO

QUICK START: RUNNING SQUAD LOCALLY

SQUAD is a Django application and works just like any other Django application. If you are new to Django and want to setup a development environment, you can follow the instructions below. If you want to install SQUAD for production usage, see *Installation Instructions for production environments* instead.

Note that SQUAD is Python3-only, so it won't work with Python 2.

Before moving on, there's a system dependency needed for Python to load yaml content with the C library, install it with:

```
apt-get install libyaml-dev
```

On top of that, the following development packages may be required. Please make sure they're installed by issuing:

```
apt-get install libpq-dev python3-dev build-essential
```

To install the dependencies:

```
pip3 install -r requirements-dev.txt
```

Alternatively to using pip, on Debian stretch or later you can install dependencies from the repository:

```
apt-get install python3-dateutil python3-django python3-celery \
  python3-django-celery python3-jinja2 python3-whitenoise python3-zmq
```

To run the tests:

```
./manage.py test
```

Before running the application, create the database and an admin user for yourself:

```
./manage.py migrate
./manage.py createsuperuser
```

To run the application locally:

```
./manage.py runserver
```

CHAPTER

THREE

DEVELOPMENT-RELATED NOTES AND TIPS

3.1 Running a development environment under Docker

To run tests, migrate database, and start the web server:

```
./dev-docker
```

To open a shell in the development environment:

```
./dev-docker bash
```

NOTE if you're running a firewall on your system, like ufw, make sure to allow the docker interface to interact with your system's. If you're running ufw, do so with *sudo ufw allow in on docker0*.

3.2 Checklist for loading a copy of a production database

This procedure assumes you are using PostgreSQL in production, and will use PostgreSQL locally. If you are using sqlite, then the procedure is trivial (just copy the database file).

on the server:

- dump the database: pg_dump -F custom -f /tmp/dump squad

locally:

- create empty DB: createdb squad
- copy dump: scp SERVER:/tmp/dump /tmp/
- load dump: pg_restore -d squad -j4 /tmp/dump
- migrate database: ./manage.py migrate
- create superuser: ./manage.py createsuperuser
- anonymize data: ./manage.py prepdump # avoid mailing users

3.3 Running Javascript unit tests

In order to run Javascript unit tests, you need to installl nodejs and npm package manager, then install the dependencies from the package-lock.json file. Depending on the distribution, you can either install npm directly from repositories or alternatevely add PPA and then install it. Here's the instructions of how to setup up after the npm package manager is installed:

```
sudo apt-get install chromium
npm install
```

Simply running the Django tests will also run the Javascript unit tests:

```
./manage test
```

Or, you can run only the Javascript unit tests with one of these commands:

```
python3 python3 test/javascript.py  # or
python3 -m test.javascript
```

3.4 Log monitoring

SQUAD uses Python's logging library to log events during its execution, it's important to keep track of those and sometimes it's nice to have an extra tool to give admins a heads up that things aren't working correctly for example when an *ERROR* log comes up.

In such scenario, SQUAD will try to send emails with the log content to admins registered in *SQUAD_ADMINS* environment variable.

SQUAD also support log monitoring and aggregation with Sentry, a tool that collects similar error logs and manage them nicer than just regular text emails. To enable Sentry support two steps are needed:

- set *SENTRY_DSN* environment variable with a dsn retrieved after creating a project in sentry.
- install Sentry's Python SDK: *pip install sentry-sdk*

CHAPTER

FOUR

TRANSLATING THE SQUAD USER INTERFACE

The SQUAD translations are kindly hosted by Weblate.

The translations are split into different components, which match the separate modules in the squad codebase. As of this writing, we have the components *core* and *frontend*. When you are reading this, we could have others.

In order to work with SQUAD translations, you need to create a weblate account.

To translate SQUAD into your language, just go to the project page on weblate, click on the component you want, and then click on your language.

If your language does not exist yet for that component, just click "Start new translation".

Translation updates made on weblate are sent back to the SQUAD source code repository once a day.

CHAPTER

FIVE

PLUGINS: USAGE AND DEVELOPMENT

5.1 Enabling plugins

Every available plugin needs to be enabled for each project in which it should be used. For that, access the Administration interface, edit the project, and add the wanted plugin names to the "Enabled plugin list" field.

5.2 Declaring plugins in your Python package

SQUAD plugins are Python classes that are a subclass of *squad.core.plugins.Plugin*, and can be provided by any Python package installed in the system. To register the plugin with SQUAD, you need to use the "entry points" system. In the setup.py for your package, use the following:

```
setup(
    # ...
    packages='mypluginpackage'
    # ...
    entry_points={
        # ...
        'squad_plugins': [
            'myplugin1=mypluginpackage.Plugin1',
            'myplugin2=mypluginpackage.Plugin2',
        ]
    },
    # ...
)
```

Now, the plugin itself can be implemented in *mypluginpackage.py*, like this:

```
from squad.core.plugins import Plugin

class Plugin1(Plugin):
    # implementation of the plugin methods ...

class Plugin2(Plugin):
    # implementation of the plugin methods ...
```

The next next section, "The plugin API" documents which methods can be defined in your plugin class in order to provide extra functionality to the SQUAD core.

5.3 The plugin API

class squad.core.plugins.**Plugin**

This can be used to pass extra arguments to plugins

extra_args = {}

This class must be used as a superclass for all SQUAD plugins. All the methods declared here have empty implementations (i.e. they do nothing), and should be overriden in your plugin to provide extra functionality to the SQUAD core.

get_url(*object_id*)

This method might return service specific URL with given object_id

has_subtasks()

This method tells whether or not the plugin will use subtasks to complete work, meaning that the main function will return but more results are still working in parallel.

notify_patch_build_created(*build*)

This method is called when a patch build is created. It should notify the corresponding patch source that the checks are in progress.

The `build` argument is an instance of `squad.core.Build`.

notify_patch_build_finished(*build*)

This method is called when a patch build is finished. It should notify the patch source about the status of the tests (success, failure, etc).

The `build` argument is an instance of `squad.core.Build`.

postprocess_testjob(*testjob*)

This method is called after a test job has been fetched by SQUAD, and the test run data (tests, metrics, metadata, logs, etc) have been saved to the database.

You can use this method to do any processing that is specific to a given CI backend (e.g. LAVA).

The `testjob` arguments is an instance of `squad.ci.models.TestJob`.

postprocess_testrun(*testrun*)

This method is called after a test run has been received by SQUAD, and the test run data (tests, metrics, metadata, logs, etc) have been saved to the database.

You can use this method to parse logs, do any special handling of metadata, test results, etc.

The `testrun` arguments is an instance of `squad.core.models.TestRun`.

5.4 Adding plugin usage to the SQUAD core

Code from the SQUAD core that wants to invoke functionality from plugins should use the `apply_plugins` function.

squad.core.plugins.**apply_plugins**(*plugin_names*)

This function should be used by code in the SQUAD core to trigger functionality from plugins.

The `plugin_names` argument is list of plugins names to be used. Most probably, you will want to pass the list of plugins enabled for a given project, e.g. `project.enabled_plugins`.

Example:

```
from squad.core.plugins import apply_plugins

# ...

for plugin in apply_plugins(project.enabled_plugins):
    plugin.method(...)
```

5.5 Full plugin package example

This section presents a minimal, working example of a Python package that provides one SQUAD plugin. It is made of only two files: `setup.py` and `examplepluginpackage/__init__.py`.

`setup.py`:

```
from setuptools import setup, find_packages

setup(
    name='examplepluginpackage',
    version='1.0',
    author='Plugin Writer',
    author_email='plugin.writer@example.com',
    url='https://example.com/examplepluginpackage',
    packages=find_packages(),
    include_package_data=True,
    entry_points={
        'squad_plugins': [
            'externalplugin=examplepluginpackage:MyPlugin',
        ]
    },
    license='GPLv3+',
    description="An example Plugin pacakge",
    long_description="""
    The Example Plugin package is a sample plugin for SQUAD that
    shows how to write SQUAD plugins
    """,
    platforms='any',
)
```

`examplepluginpackage/__init__.py`:

```
from squad.core.plugins import Plugin

class MyPlugin(Plugin):

    def postprocess_testrun(self, testrun):
    # do something interesting with the the testrun ...
        pass
```

5.6 Built-in notification plugins

SQUAD comes with two bult-in plugins available for immediate use.

5.6.1 Github

The Github plugin allows patches (Pull Requests) originated from Github to be notified whenever a build has been created or finished.

Here is an example API call that supposedly came from a Jenkins job, triggered by a freshly opened Github Pull Request:

```
$ curl \
    -X POST \
    --header "Authorization: token $SQUAD_TOKEN" \
    -d patch_source=your-github-patch-source \
    -d patch_baseline=build-v1 \
    -d patch_id=the_owner/the_repo/8223a534d7bf \
    https://squad.example.com/api/createbuild/my-group/my-project/build-v2
```

Where:

- *patch_source* is the name of a "Patch Source" previously added in squad in "Administration > Core > Patch sources > Add patch source", where you should select "github" for "implementation". **NOTE** the Github plugin requires a *token* for authentication, so please ignore the "password" field.
- *patch_baseline* is an optional parameter that indicated that the build being created is a new version of "patch_baseline" build.
- *patch_id* is a string in a form like "owner/repository/commit" of the respective Github repository.

If everything was successfully submitted, you should see a notification in the Github page for that Pull Request. Subsequent tests on that build are going to be performed and as SQUAD detects that all tests are done, another notification should be sent out on that Pull Request, informing that the build is finished.

5.6.2 Gerrit

The Gerrit plugin allows changes originated from a Gerrit instance to be notified whenever a build has been created or finished.

Here is an example API call that supposedly came from a Jenkins job, triggered by a freshly created change:

```
$ curl \
    -X POST \
    --header "Authorization: token $SQUAD_TOKEN" \
    -d patch_source=your-gerrit-patch-source \
    -d patch_baseline=build-v1 \
    -d patch_id=change-id/patchset \
    https://squad.example.com/api/createbuild/my-group/my-project/build-v2
```

Where:

- *patch_source* is the name of a "Patch Source" previously added in squad in "Administration > Core > Patch sources > Add patch source", where you should select "gerrit" for "implementation". **NOTE 1** the Gerrit plugin requires a *password* (configured as HTTP Password in Gerrit) for authentication, so please ignore the "token" field. **NOTE 2** the Gerrit plugin also allows SSH based notifications by using "ssh://" instead

of "https://" in the "url" field. **NOTE 3** SSH connections are made only through key exchange, so please set it up before attempting to use this feature

- *patch_baseline* is an optional parameter that indicated that the build being created is a new version of "patch_baseline" build.
- *patch_id* is a string in a form like "change-id/patchset" of the respective Gerrit repository.

NOTE

By default, the plugin will only apply "Code-Review -1" for builds that errored. Custom labels are supported if specified in project settings. Here is a example on how to specify custom labels for gerrit:

```
plugins:
  gerrit:
    build_finished:
      success:
        Code-Review: "+1"
        Validation-Bot-Review: "+1"
      error:
        Code-Review: "-1"
        My-Custom-Bot-Review: "-1"
```

If everything was successfully submitted, you should see a notification in the Gerrit page for that Change. Subsequent tests on that build are going to be performed and as SQUAD detects that all tests are done, another notification should be sent out on that Change, informing that the build is finished.

CHAPTER

SIX

INSTALLATION INSTRUCTIONS FOR PRODUCTION ENVIRONMENTS

6.1 Installation using the Python package manager pip

Make sure you have the `pip` Python package manager installed on your Python 3 environment, and the C library for YAML development. On Debian/Ubuntu, the easiest way to do that is:

```
apt-get install python3-pip libyaml-dev
```

Install squad:

```
pip3 install squad
```

By default, SQUAD works with sqlite, but if Postgres is required, then specific binaries are needed:

```
apt-get install python3-pip libyaml-dev libpq-dev
```

Install squad with Postgres support:

```
pip3 install squad[postgres]
```

6.2 Message broker

In order to SQUAD processes to be able to communicate between each other, you need to install an AMQP server. We recommend RabbitMQ:

```
apt-get install rabbitmq-server
```

By default SQUAD will look for an AMQP server running on localhost, listening to the standard port, so for a single-server deployment you don't need to do anything else.

If you have a multi-server setup, then each server needs to be configured with the location of a central AMQP server. See the *SQUAD_CELERY_BROKER_URL* in the "Further configuration" section below.

6.3 Processes

SQUAD is composed of 4 different process:

- web application server
- background worker (celery worker)
- periodic task scheduler (celery beat)
- CI backend listener

To run the web interface, run as a dedicated user (i.e. don't run the application as `root`!):

```
squad
```

- Note: if that doesn't work, `~/.local/bin` is probably missing in the `$PATH` environment variable.

This will make the web UI available at http://localhost:8000/. To serve the UI to external users, you will need to setup a public-facing web server such as Apache or nginx and reverse proxy to localhost on port 8000. You can change the address or port SQUAD listens to by passing the `--bind` command line option, e.g. to make it listen to port 5000 on the local loopback interface, use:

```
squad --bind 127.0.0.0:5000
```

For production usage, you will want to tweak at least the database configuration. Keep reading for more information.

After starting SQUAD, but before acessing it, you will need a user. To create an admin user for yourself, use:

```
squad-admin createsuperuser
```

These are the command lines to run the other processes:

Process	Command
worker	celery -A squad worker
scheduler	celery -A squad beat
listener	squad-admin listen

You most probably want all the processes (including the web interface) being managed by a system manager such as systemd, or a process manager such as supervisor.

For an example deployment, check the configuration management repository for Linaro's qa-reports (using ansible).

After having the necessary processes running, there are a few extra setup steps needed:

- Create `Backend` instances for your test execution backends. Go to the administration web UI, and under "CI", choose "Backends".
- For each project, create authentication tokens and subscriptions

6.4 Worker configuration

The *worker* process handles background tasks, such as submitting CI jobs, fetching the results fo CI jobs, preparing reports that require intensive processing, etc. Some tasks take a lot longer than the others, e.g. submitting a CI job is pretty quick, while fetching CI job results takes some time to fetch all the data, parse it, and store in the database.

To allow for better load balancing, these tasks are split into multiple queues:

- *ci_fetch*
- *ci_poll*
- *ci_quick*
- *core_notification*
- *core_postprocess*
- *core_quick*
- *core_reporting*

ci_fetch and *ci_poll* can be potentially slow, and if there is a large influx of those types of tasks, your system may display some congestion because all available worker threads are occupied running slow tasks, while several of the quick ones are waiting their turn.

To avoid this, you might want to start a small part of your workers so that they will not pick up any of those slow tasks:

```
squad worker --exclude-queues ci_fetch,ci_poll
```

Or you can also give an explicit task list, which is less flexible but might be useful:

```
squad worker --queues core_quick,ci_quick
```

By default, workers listen to all queues.

For message brokers that support prefixed-queue names, SQUAD has the optional environment variable *SQUAD_CELERY_QUEUE_NAME_PREFIX*, that, if set, will prepend it before all queue names. SQUAD also support adding a suffix via the optional environment variable *SQUAD_CELERY_QUEUE_NAME_SUFFIX*

6.5 Further configuration

The following environment variables affect the behavior of SQUAD:

- `DATABASE`: controls the database connection parameters. Should contain `KEY=VALUE` pairs, separated by colons (`:`).

  By default, SQUAD will use a SQLite3 database in its internal data directory.

  For example, to use a PostgreSQL database (requires the `psycopg2` Python package to be installed):

  ```
  DATABASE=ENGINE=django.db.backends.postgresql_
  ↪psycopg2:NAME=mydatabase:USER=myuser:HOST=myserver:PASSWORD=mypassword
  ```

- `SQUAD_EXTRA_SETTINGS`: path to a Python file with extra Django settings.
- `SQUAD_SITE_NAME`: name to be displayed at the page title and navigation bar. Defaults to 'SQUAD'.
- `XDG_DATA_HOME`: parent directory of the SQUAD internal data directory. Defaults to `~/.local/share`. The actual data directory will be `${XDG_DATA_HOME}/squad`.

- SECRET_KEY_FILE: file to store encryption key for user sessions. Defaults to ${XDG_DATA_HOME}/squad/secret.dat
- DJANGO_LOG_LEVEL: the logging level used for Django-related logging. Default: INFO.
- SQUAD_LOG_LEVEL: the logging level for SQUAD-specific logging. Default: INFO.
- SQUAD_HOSTNAME: hostname used to compose links in asynchronous notifications (e.g. emails). Defaults to the FQDN of the host where SQUAD is running.
- SQUAD_BASE_URL: Base URL to the SQUAD web interface, used when composing links in notifications (e.g. emails). Defaults to https://$SQUAD_HOSTNAME.
- SQUAD_EMAIL_FROM: e-mail used as sender of email notifications. Defaults to noreply@$SQUAD_HOSTNAME.
- SQUAD_EMAIL_HOST: hostname to use as e-mail delivery host. Sets Django's EMAIL_HOST setting. See the Django documentation on sending email for more details.
- SQUAD_LOGIN_MESSAGE: a message to be displayed to users right above the login form. Use for example to provide instructions on what credentials to use. Defaults no message.
- SQUAD_ADMINS: Comma-separated list of administrator email addresses, for use in exception notifications. Each address must be formatted as First Last <first.last@example.com>.
- SQUAD_SEND_ADMIN_ERROR_EMAIL: Determines whether or not to send exception notifications to administrators. Defaults to True.
- SENTRY_DSN: Defines Sentry's DSN token, if defined SQUAD will attempt to import Sentry SDK and use it. Defaults to None. If Sentry is configured it's recommended to disable sending admin notifications by setting SQUAD_SEND_ADMIN_ERROR_EMAIL = False.
- SQUAD_STATIC_DIR: Directory where SQUAD will find it's preprocessed static assets. This usually does not need to be set manually, and exists mostly for use in the Docker image.
- SQUAD_CELERY_BROKER_URL: URL to the broker to be used by Celery for background jobs. Defaults to amqp://localhost:5672.
- SQUAD_CELERY_QUEUE_NAME_PREFIX: Name to prefix all queues in Celery. Useful when multiple environments share the same broker. Defaults to ''.
- SQUAD_CELERY_QUEUE_NAME_SUFFIX: Name to concatenate all queues in Celery. Useful when a queue extension is needed by the broker. Defaults to ''.
- SQUAD_CELERY_POLL_INTERVAL: Number of seconds a worker will sleep after an empty answer from SQS before the next polling attempt. Defaults to 1.

6.6 User management

SQUAD provides 'users' management command that allows to list, add, update and display details about users. This command comes handy when trying to automate SQUAD setup with containers. Details about user management with 'users' command:

- list Displays list of all available users with their names (first, last) from database
- details <username> Displays details about requested username. Details include:
  - username
  - is_active
  - is_staff
  - is_superuser

  - groups
- add <username> Adds new user with given 'username'. It also takes additional parameters
  - --email EMAIL email of the user
  - --passwd PASSWD Password for this user. If empty, a random password is generated.
  - --staff Make this user a staff member
  - --superuser Make this user a super user
- update <username> Updates database record of existing user identified with 'username'. It takes additional parameters
  - --email EMAIL Change email of the user
  - --active Make this user active
  - --not-active Make this user inactive
  - --staff Make this user a staff member
  - --not-staff Make this user no longer a staff member
  - --superuser Make this user a superuser
  - --not-superuser Make this user no longer a superuser

6.7 Docker compose setup

Alternatively, SQUAD can be run inside a Docker container using Docker Compose. A baseline setup is provided here.

Not only will this run all SQUAD processes, but also automatically pull and start all required services to run a local SQUAD instance with a web interface. All container images are pulled automatically from Dockerhub.

Please be aware, that some of the predefined settings, like `SQUAD_BASE_URL` or `SQUAD_EMAIL_FROM` in `docker-compose.yml` and `squad-apache.conf` will need to be changed. To use this setup, copy all of the provided files locally and run the docker using:

```
docker compose up -d
```

After that, the logs and status of the container can be checked resp. using:

```
docker compose logs
```

and:

```
docker stats
```

If everything has started successfully, the frontend should be reachable under http://localhost:10080.

Before being able to actually use the frontend of SQUAD, a user that acts as an administrator (with superuser rights) needs to be created. This can be done by entering the container with:

```
docker compose exec squad-frontend /bin/bash
```

While being in the container, a new administrator account can be created with the following command:

```
squad-admin createsuperuser
```

After this command, enter the required data and exit the container environment by issuing `exit`.

Now, you should be able to login on the frontend web interface, using the credentials you just entered while creating the administrator account.

To stop the SQUAD instance, issue the following command to stop the container:

```
docker compose down
```

6.7.1 Database dump and restore

Note that container must be running for the following commands to work.

To dump the database to the file *db-dumped*, issue the following command:

```
docker compose exec -t postgres pg_dump -F custom -U postgres -f /db-dumped squad
```

To restore the database from the file *db-to-be-restored*, issue the following command:

```
docker compose exec -t postgres pg_restore -U postgres -c -d squad -j4 /db-to-be-restored
```

CHAPTER

SEVEN

CI: CONTINOUS INTEGRATION SUPPORT

7.1 CI module in SQUAD

This subsystem has the following features:

- receiving test job requests
- submitting test job requests to test execution backends
- pulling test job results from test execution backends

The data model for the CI subsystem looks like this:

```
+---------+    +---------+    +------------------------+
| TestJob |--->| Backend |--->| Backend implementation |
+---------+    +---------+    +------------------------+
     |
     |         +----------------------+
     +-------->| TestRun (from core) |
               +----------------------+
```

TestJob holds the data related to a test job request. This test job is going to be submitted to a Backend, and after SQUAD gets results back from that backend, it will create a TestRun object with the results data. A Backend is a representation of a given test execution system, such as a LAVA server, or Jenkins. `Backend` contains the necessary data to access the backend, such as URL, username and password, etc, while `Backend implementation` encapsulates the details on how to interact with that type of system (e.g. API calls, etc). So for example you can have multiple backends of the same type (e.g. different 2 LAVA servers).

For the CI loop integration to work, you need to run a few extra processes beyond the web interface. See *Installation Instructions for production environments* for details.

7.2 Submitting test job requests

The API is the following

POST /api/submitjob/:group/:project/:build/:environment

- `group`, `project`, `build` and `environment` are used to identify which project/build/environment will be used to record the results of the test job.
- The following data must be submitted as POST parameters:
  - `backend`: name of a registered backend, to which this test job will be submitted.

  - `definition`: test job definition. The contents and format are backend-specific. If it is more convenient, the definition can also be submitted as a file upload instead of as a POST parameter.

Example (with test job definition as POST parameter):

```
$ DEFINITION="$(cat /path/to/definition.txt)"
$ curl \
    --header "Authorization: token $SQUAD_TOKEN" \
    --form backend=lava \
    --form definition="$DEFINITION" \
    https://squad.example.com/api/submitjob/my-group/my-project/x.y.z/my-ci-env
```

Example (with test job definition as file upload):

```
$ curl \
    --header "Authorization: token $SQUAD_TOKEN" \
    --form backend=lava \
    --form definition=@/path/to/definition.txt \
    https://squad.example.com/api/submitjob/my-group/my-project/x.y.z/my-ci-env
```

Submitted jobs on finished builds do not cause events (email/patchsource/callback notifications) to be triggered when the job is fetched. But there's an option named *CI_RESET_BUILD_EVENTS_ON_JOB_RESUBMISSION* that tells SQUAD to reset all build events on job submission so that they can be triggered once more next time the build reaches its "finished" state.

The user owning the SQUAD_TOKEN should be a member of the group and should have the "Staff Status" permission.

7.3 Submitting test job watch requests

Test job watch request are similar to test job requests. The only difference is that some other service submitted the test job for execution and SQUAD is requested to track the progress. By default, SQUAD will schedule the job for fetching right away. If the variable *?delay_fetch* is present, SQUAD will wait until the test job is finished before retrieving the results and do post processing. The API is following:

POST /api/watchjob/:group/:project/:build/:environment

- `group`, `project`, `build` and `environment` are used to identify which project/build/environment will be used to record the results of the test job.
- The following data must be submitted as POST parameters:
  - `backend`: name of a registered backend, to which this test job was be submitted.
  - `testjob_id`: test job ID. The contents and format are backend-specific.

Example (with test job definition as POST parameter):

```
$ curl \
    --header "Authorization: token $SQUAD_TOKEN" \
    --form backend=lava \
    --form testjob_id=123456 \
    https://squad.example.com/api/watchjob/my-group/my-project/x.y.z/my-ci-env
```

7.4 Backend settings

Backends support internal settings that are stored in the database. It is assumed that settings are a valid YAML markup.

7.5 Supported backends

Out of the box SQUAD supports following backends:

- LAVA
- TuxSuite

7.5.1 LAVA

SQUAD supports only LAVA v2. Old version of LAVA was made obsolete with 2017.11 LAVA release.

LAVA backend supports the following settings:

- CI_LAVA_INFRA_ERROR_MESSAGES a list of strings that cause automated job resubmission when matched in the LAVA error message
- CI_LAVA_SEND_ADMIN_EMAIL boolean flag that prevents sending admin emails for each resubmitted job when set to `False`
- CI_LAVA_HANDLE_SUITE boolean flag that parses results from LAVA test suite when set to `True`. Please note that this option can be overwritten by having the same option with different value in Project *project_settings*
- CI_LAVA_CLONE_MEASUREMENTS boolean flag that allows to save LAVA result as both Test and Measurement when set to `True`. Default is `False`. Can be overwritten for each project separately (similar to CI_LAVA_HANDLE_SUITE).
- CI_LAVA_HANDLE_BOOT boolean flag that parses LAVA *auto-login-action* as a boot test when set to `True`. Default is `False`. Can be overwritten for each project separately (similar to CI_LAVA_HANDLE_SUITE). **NOTE**: Before SQUAD 1.x series, the default behavior was to always process *auto-login-action* as boot. After 1.x, the default behavior has changed to do the opposite.
- CI_LAVA_WORK_AROUND_INFRA_ERRORS boolean flag that allows to accept test results from 'Incomplete' jobs if the failure was caused by infrastracture. **NOTE**: Use with caution!
- CI_LAVA_JOB_ERROR_STATUS string that coincides with the LAVA job health. Used when sending email notifications for the ON_ERROR notification strategy

Example LAVA backend settings:

```
CI_LAVA_INFRA_ERROR_MESSAGES:
  - 'Connection closed'
  - 'lava_test_shell connection dropped.'
  - 'fastboot-flash-action timed out'
  - 'u-boot-interrupt timed out'
  - 'enter-vexpress-mcc timed out'
  - 'Unable to fetch git repository'
CI_LAVA_SEND_ADMIN_EMAIL: False
CI_LAVA_HANDLE_SUITE: True
```

7.5.2 TuxSuite

SQUAD supports only LAVA v2. Old version of LAVA was made obsolete with 2017.11 LAVA release.

LAVA backend supports the following settings:

- TEST_METADATA_KEYS a list of strings to extract metadata info from test results
- BUILD_METADATA_KEYS a list of strings to extract metadata info from build results

Example TuxSuite backend settings:

```
BUILD_METADATA_KEYS:
  - 'build_status'
  - 'download_url'
  - 'git_describe'
  - 'git_ref'
  - 'git_repo'
  - 'git_sha'
  - 'git_short_log'
  - 'kernel_version'
  - 'kconfig'
  - 'target_arch'
  - 'toolchain'
```

Multinode

SQUAD supports fetching results from LAVA multinode jobs. There are however a few limitations with this setup:

- All results from multinode will share environment name Since test jobs are submitted via SQUAD using the environment from submit URL there is no way for SQUAD to distinguish between different environmens on different parts of multinode job.
- Resubmit will repeat the whole set In SQUAD all parts of multinode job will share the multinode definition. For this reason re-submitting any part of the multinode job will result in new multinode job that includes all parts.
- Each part of the multinode job will be retrieved separately This means that each part will create a TestRun in SQUAD. This should not be a major issue as all results will still be available. Users need to make sure that the test names don't overlap as SQUAD will not have any means of distinguishing between identically named tests from different parts of multinode job.

7.6 Callbacks Triggers

In SQUAD, callbacks can be attached to Builds. They are triggered once the given build finishes fetching all test jobs from the backend.

There's currently two ways of attaching a callback to a build:

- *POST /api/createbuild/<group_slug>/<project_slug>/<build_version>* (when creating a build)
- *POST /api/build/<build_id>/callbacks/* (attach to an existing build)

And the following parameters are accepted for both endpoints:

```
$ curl -X POST /api/build/<build_id>/callbacks/ \
       -F "callback_url=https://your-callback-url.com"
```

The following attributes are optional:

- callback_method - string `post` or `get` defining the callback request method. Defaults to `post`
- callback_event - string `on_build_finished` defining at which point the callback should be dispatched. Defaults to `on_build_finished`
- callback_headers - JSON-formatted string defining the callback headers, useful to define auth tokens
- callback_payload - JSON-formatted string defining the callback payload
- callback_payload_is_json - string with `true` or `false` indicating whether the payload should be sent as JSON or as form-data. Defaults to `true`
- callback_record_response - string with `true` or `false` indicating whether or not the callback response should be recorded in SQUAD

7.6.1 Authentication

Callbacks usually require some sort of authentication. In SQUAD this can be accomplished in two forms:

- via `callback_headers`, where a JSON-formatted string is expected and will be used in the callback headers when it triggers. Ex:

```
$ curl -X POST /api/build/<build_id>/callbacks/ \
       -F "callback_url=https://your-callback-url.com" \
       -F "callback_headers='{\"Auth-Token\": \"your-really-safe-token\"}'"
```

- via project settings `/<group_slug>/<project_slug>/settings/advanced/` (YAML-formatted):

```
CALLBACK_HEADERS:
  Auth-Token: your-really-safe-token
```

7.6.2 Notes

It's important to point out that:

- Multiple callbacks are allowed for a build, given that they point to different urls
- Attaching the same callback twice to the same build results in noop
- The callback headers will be merged with the build's project settings if available. If header names collide, project settings will get overwritten
- Callbacks are available in read-only mode at `GET /api/builds/<id>/callbacks/`

7.7 Receiving Callbacks

SQUAD also supports receiving callbacks. Currently the use case for receiving callbacks is for backends that need to push results back to SQUAD while not having a live connection like LAVA does with ZMQ or websockets. Tuxsuite is a good example as it runs on a serverless architecture it doesn't provide live connections, thus needing the callback feature.

The callback URL should be in format:

- *POST /api/fetchjob/<group_slug>/<project_slug>/<build_version>/<environment-slug>/<backend-name>*

Authetication and payloads are dependant on the backend implementation. There is currently only one supported backend: Tuxsuite. In the section below we will describe how this integration should work.

7.7.1 Use case: Tuxsuite

SQUAD allows callbacks to be triggered by Tuxsuite. Developers trigger builds and tests to Tuxsuite as they would normally do. The difference now is that they can pass a URL to be POST'ed after such build or test is finished. Below is an example of how to do that:

```
$ tuxsuite build \
    --git-repo https://github.com/torvalds/linux.git \
    --git-ref master \
    --target-arch arm \
    --toolchain gcc-12 \
    --kconfig tinyconfig \
    --callback https://squad.com/api/fetchjob/tuxgroup/tuxproject/mybuild/myenv/tuxsuite
```

This tells Tuxsuite to POST to *https://squad.com/api/fetchjob/tuxgroup/tuxproject/mybuild/myenv/tuxsuite*. Payload and authentication are Tuxsuite-specific and documentation can be found at https://docs.tuxsuite.com/callbacks/.

In order to validate that the request is coming from Tuxsuite, SQUAD checks the *x-tux-payload-signature* header and match it with public key configured in each project setting page.

SQUAD will attempt to read key from *TUXSUITE_PUBLIC_KEY* variable defined in the project settings of *tuxgroup/tuxproject*. If the request is valid, SQUAD will take in the payload provided by Tuxsuite, save it and enqueue a test job for fetching.

The main difference now is that Tuxsuite will be the one telling SQUAD when to fetch results. This prevents SQUAD from polling Tuxsuite every now and then.

CHAPTER

EIGHT

API: INTERACTING WITH BACKEND

8.1 Available APIs

SQUAD has a set of APIs that allow to interact with it's backend. There are two main parts of the API

- Native API This is meant to provide main features of SQUAD (submitting results, submitting CI test jobs)
- REST API Provides access to almost all properties of core data model objects. Also provides additional features that can be used to build alternative frontends or automated tools.

8.2 Native APIs

8.2.1 data

GET /api/data/<group_slug>/<project_slug>/

Retrieves metrics data in JSON format. The following parameters are mandatory:

- *metric*: which metric to retrieve. You have to use the full metric name, i.e. *<suite_slug>/<metric_slug>*.

  This parameter can be specified multiple times, so data from multiple metrics can be fetched with a single request.

- *environment*: environment for which metric data is to be retrieved.

  This parameter can be specified multiple times, so data from multiple environments can be fetched with a single request.

- *format*: format of response. Valid values are *json* and *csv*. If this parameter is ommited, *json* is used as a default.

The JSON response is an object, which metrics as keys. Values are also objects, which environments as keys, and the data series as values. Each data point is an array with 3 values: the build date timestamp (as the number of seconds since the epoch), the value of the metric, and the build identifier.

Example:

```
{
    "mysuite/mymetric": {
        "environment1": [
            [1537210872, 1.15, "v0.50.1-21-g7b96236"],
            [1537290845, 1.14, "v0.50.1-22-g1097312"],
            [1537370812, 1.13, "v0.50.1-23-g0127321"],
            [1537420892, 1.15, "v0.50.1-24-g8262524"],
            [1537500801, 1.13, "v0.50.1-25-gfa72526"],
            // [...]
```

(continues on next page)

(continued from previous page)

```
        ],
        "environment2": [
            [1537210872, 1.25, "v0.50.1-21-g7b96236"],
            [1537290845, 1.24, "v0.50.1-22-g1097312"],
            [1537370812, 1.23, "v0.50.1-23-g0127321"],
            [1537420892, 1.25, "v0.50.1-24-g8262524"],
            [1537500801, 1.23, "v0.50.1-25-gfa72526"],
            // ...
        ]
    },
    "mysuite/anothermetric": {
        // [...]
    }
}
```

The CSV response contains one line for each data point. The columns are: metric, environment, timestamp, value, build identifier. Assuming the same data as the JSON example above, the CSV would look like this:

```
"mysuite/mymetric","environment1","1537210872","1.15","v0.50.1-21-g7b96236"
"mysuite/mymetric","environment1","1537290845","1.14","v0.50.1-22-g1097312"
"mysuite/mymetric","environment1","1537370812","1.13","v0.50.1-23-g0127321"
"mysuite/mymetric","environment1","1537420892","1.15","v0.50.1-24-g8262524"
"mysuite/mymetric","environment1","1537500801","1.13","v0.50.1-25-gfa72526"
[...]
"mysuite/mymetric","environment2","1537210872","1.25","v0.50.1-21-g7b96236"
"mysuite/mymetric","environment2","1537290845","1.24","v0.50.1-22-g1097312"
"mysuite/mymetric","environment2","1537370812","1.23","v0.50.1-23-g0127321"
"mysuite/mymetric","environment2","1537420892","1.25","v0.50.1-24-g8262524"
"mysuite/mymetric","environment2","1537500801","1.23","v0.50.1-25-gfa72526"
[...]
"mysuite/anothermetric",[...]
[...]
```

8.2.2 createbuild

POST /api/createbuild/<group_slug>/<project_slug>/<version_string>

Creates Build object. Following parameters are accepted:

- patch_source - string matching PatchSource.name
- patch_baseline - version string matching Build.version
- patch_id - string identifying the patched version (for example git commit ID)
- callback_url - url string to define a callback for the build
- callback_method - string "post" or "get" defining the callback request method. Defaults to "post"
- callback_event - string "on_build_finished" defining at which point the callback should be dispatched. Defaults to "on_build_finished"
- callback_headers - JSON-formatted string defining the callback headers, useful to define auth tokens
- callback_payload - JSON-formatted string defining the callback payload

- callback_payload_is_json - string with "true" or "false" indicating whether the payload should be sent as JSON or as form-data. Defaults to "true"
- callback_record_response - string with "true" or "false" indicating whether or not the callback response should be recorded in SQUAD

8.2.3 submit

Submitting results.

8.2.4 submitjob

Submitting test job requests.

8.2.5 watchjob

Submitting test job watch requests.

8.2.6 resubmit

POST /api/resubmit/<job_id>

This API is only available to superusers at the moment. It allows to resubmit CI test jobs using Backend's implementation.

By default, the original test job object's results are kept in, but there's a project setting named *CI_DELETE_RESULTS_RESUBMITTED_JOBS* that tells SQUAD to remove all results from the resubmitted job.

Resubmitted jobs on finished builds do not cause events (email/patchsource/callback notifications) to be triggered when the job is fetched. But there's an option named *CI_RESET_BUILD_EVENTS_ON_JOB_RESUBMISSION* that tells SQUAD to reset all build events on job resubmission so that they can be triggered once more next time the build reaches its "finished" state.

8.2.7 forceresubmit

POST /api/forceresubmit/<job_id>

This API is only available to superusers at the moment. It allows to resubmit CI test jobs using Backend's implementation. Works similarly to 'resubmit' but doesn't respect 'can_resubmit' flag on the TestJob object.

8.3 REST APIs

The REST API is powered by **`Django Rest Framework (DRF)<https://www.django-rest-framework.org/>`_** and Django fields lookups. This means that for supported endpoints you can do a field lookup. For example, querying all testruns that belong to a build that belongs to a project called MyProject, one would run a query like:

GET /api/testruns/?build__project__name=MyProject

This gives the API flexibility for filtering in many different ways.

8.3.1 groups (/api/groups/)

Provides access to Group object. This object corresponds to SQUAD Group (not to be confused with Django group). The Group objects can be filtered and searched. Both operations can be done using 'name' and 'slug' fields.

With enough privileges Groups can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.2 projects (/api/projects/)

Provides access to Project object. In case of private projects token with enough privileges is required to access the object. Project API endpoint has following additional routes:

- builds (/api/projects/<id>/builds/)

  Provides list of builds associated with this project. List is paginated

- test_results (/api/projects/<id>/test_results/)

  Provides list of latest results for given test for all environments. 'test_name' is a mandatory GET parameter for this call. List is paginated. It is advised to limit the search results to 10 to avoid poor performance. This can be achieved using 'limit=10' GET parameter

- subscribe (/api/projects/<id>/subscribe/)

  Provides means to subscribe either email address or user to the project notifications in automated way. This endpoint expects POST request with single field "email"

- unsubscribe (/api/projects/<id>/unsubscribe/)

  Provides means to unsubscribe either email address or user from the project notifications in automated way. This endpoint expects POST request with single field "email"

With enough privileges Projects can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.3 builds (/api/builds/)

Provides access to Build object. In case of private projects token with enough privileges is required to access the object. Build API endpoint has following additional routes:

- metadata (/api/builds/<id>/metadata/)

  Provides list of all metadata key-value pairs associated with this object

- status (/api/builds/<id>/status/)

  Provides access to ProjectStatus object associated with this object

- testruns (/api/builds/<id>/testruns)

  Provides list of TestRun objects associated with this object

- testjobs (/api/builds/<id>/testjobs/)

  Provides list of TestJob objects associated with this object

- email (/api/builds/<id>/email/)

  Provides contents of email notification that would be generated for this object. Content is generated using either EmailTemplate associated with the Project or a custom one. The EmailTemplate has to be defined in SQUAD database before API is called. The route takes the following GET parameters:

  - output - mime type to be generated. Defaults to "text/plain". Can also be set to "text/html". Using HTML requires HTML part of the EmailTemplate to be defined
  - template - ID of the EmailTemplate to be used
  - baseline - ID of the Build object to be used as comparison baseline. The default is "previous finished" build in the same project.
  - force - if set to true invalidates cached object. Default is false

- report (/api/build/<id>/report/)

  This API accepts both GET and POST requests.

  Provides non blocking version of 'email' API. Both calls will produce DelayedReport objects which cache the results of the call. Non blocking version ('report') is recommended as it is executed in separate process on the worker node and doesn't affect web frontend performance or memory consumption. Reports might be resource hungry and long running which causes webserver requests to time out. Non blocking call returns immediately returning url to the cached resource. Final results can be retrieved by:

  - email notification
  - callback notification
  - polling the result URL - Results are completed when 'status_code' field is filled in (not None/Null)

  'report' API has following options:

  - output - mime type to be generated. Defaults to "text/plain". Can also be set to "text/html". Using HTML requires HTML part of the EmailTemplate to be defined
  - template - ID of the EmailTemplate to be used
  - baseline - ID of the Build object to be used as comparison baseline. The default is "previous finished" build in the same project.
  - email_recipient - email address which is notified when report is ready
  - callback - URL which SQUAD calls when report is ready. Call is made using POST request type. Call can be secured with token
  - callback_token - token/password for securing callback. When "callback" option is present it adds "Authorization" and "Auth-Token" headers to the HTTP POST call. It is recommended to send this option usig POST request to avoid password leakage.
  - keep - number of days to keep the cached reports in the database
  - force - if set to true invalidates cached object. Default is false

- callbacks (/api/builds/<id>/callbacks/)

  This API accepts both GET and POST requests. On GET requests, a list of callbacks is retrieved. A POST request will create a callback and attach to this build. The following parameters are accepted:

  - callback_url - url string to define a callback for the build (the only *mandatory* field)
  - callback_method - string "post" or "get" defining the callback request method. Defaults to "post"
  - callback_event - string "on_build_finished" defining at which point the callback should be dispatched. Defaults to "on_build_finished"
  - callback_headers - JSON-formatted string defining the callback headers, useful to define auth tokens
  - callback_payload - JSON-formatted string defining the callback payload
  - callback_payload_is_json - string with "true" or "false" indicating whether the payload should be sent as JSON or as form-data. Defaults to "true"

  - callback_record_response - string with "true" or "false" indicating whether or not the callback response should be recorded in SQUAD

With enough privileges Builds can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively. This is however not recommended.

8.3.4 testjobs (/api/testjobs/)

Provides access to TestJob object. In case of private projects token with enough privileges is required to access the object. Build API endpoint has following additional routes:

- definition

  Returns plain text version of the TestJob.definition field. This is pretty specific to LAVA but doesn't exclude any other automated execution tools.

8.3.5 testruns (/api/testruns/)

Provides access to TestRun object. In case of private projects token with enough privileges is required to access the object. Build API endpoint has following additional routes:

- tests_file (/api/testruns/<id>/tests_file/)
- metrics_file (/api/testruns/<id>/metrics_file/)
- metadata_file (/api/testruns/<id>/metadata_file/)
- log_file (/api/testruns/<id>/log_file/)
- tests (/api/testruns/<id>/tests/)
- metrics (/api/testruns/<id>/metrics/)
- status (/api/testruns/<id>/status/)

  Provides a list of TestRun's statuses. One can also passing in filters to get specific results, e.g. /api/testruns/<id>/status/?suite__isnull=true retrieves the overall Status object for that testrun.

8.3.6 tests (/api/tests/)

Provides access to Tests objects. In case of private projects token with enough privileges is required to access the objects.

8.3.7 metrics (/api/metrics/)

Provides access to Metrics objects. In case of private projects token with enough privileges is required to access the objects.

8.3.8 suites (/api/suites/)

Provides access to Suite object. In case of private projects token with enough privileges is required to access the object.

8.3.9 environments (/api/environments/)

Provides access to Environment object. In case of private projects token with enough privileges is required to access the object.

8.3.10 backends (/api/backends/)

Provides access to Backend object.

With enough privileges Backend can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.11 emailtemplates (/api/emailtemplates/)

Provides access to EmailTemplate object.

With enough privileges EmailTemplate can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.12 knownissues (/api/knownissues/)

Provides access to KnownIssue object.

With enough privileges KnownIssue can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.13 patchsources (/api/patchsources/)

Provides access to PatchSource object.

8.3.14 annotations (/api/annotations/)

Provides access to Annotation object.

With enough privileges Annotation can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.15 metricthresholds (/api/metricthresholds/)

Provides access to MetricThreshold object.

With enough privileges MetricThreshold can also be created, modified and deleted using REST API with POST, PUT and DELETE HTTP requests respectively

8.3.16 reports (/api/reports/)

Provides access to results of /api/build/<id>/email and /api/build/<id>/report results. Both of these endpoints create DelayedReport objects and present them to the user. The difference is that 'email' API is blocking and 'report' is not blocking (returns immediately).

status_code field in the reports endpoint will indicate whether the report is ready. If the field is empty, the report wasn't prepared yet. status_code follows the HTTP status codes. Anything else that 200 in status_code field suggests a problem. error_message field can be checked to learn about issue details.

8.4 REST API Schema (for CLI)

SQUAD's API supports API clients. Example is coreapi. In order for client to understand the API SQUAD generates schema file. Schema is dynamically built and it's available at /api/schema URL. Example usage with coreapi-cli:

```
coreapi get https://<host_tld>/api/schema
coreapi action projects list
```

More details about coreapi can be found on coreapi website and DRF website:

- http://www.coreapi.org/
- https://www.django-rest-framework.org/topics/api-clients/

8.5 SQUAD-Client

SQUAD team has been working on a client tool that help users query the API easily, using a Python descriptive way of interacting with the backend.

If you are interested in using such tool, please check it out in SQUAD-Client

8.6 Badges

SQUAD offers project and build badges that can be used in the webpages

```
https://<squad_instance_tld>/group/project/badge
https://<squad_instance_tld>/group/project/build_version/badge
```

The colour of the badge matches the passed/failed condition. Following colours are presented:

- green (#5cb85c) when there are no failed results
- orange (#f0ad4e) when there are both passed and failed results
- red (#d9534f) when there are no passed results

If there are no results, the badge colour is grey (#999)

Badge offers customization through following parameters:

- title

  Changes the left part of the badge to a custom text

- passrate

  Changes the right part of the badge to use pass rate rather than number of tests passed, failed and skipped

- metrics

  Changes the right part of the badge to use metrics instead of test results. In such case badge colour is set to green. In case both 'metrics' and 'passrate' keywords are present, 'metrics' is ignored.

- suite/environment (only available in build badges)

  Filter results by specific suite and environment.

- hide_zeros

  When set to 1 or true, avoids printing status with zero as number. Exemple: without "hide_zeros" a badge might be like "pass: 1, fail: 0, xfail: 0", whereas if enabled, it would display the badge like "pass: 1".

8.7 Google Data Studio

SQUAD has an implementation of the Google Data Studio Community Connector under https://github.com/Linaro/squad/tree/master/scripts/community_connector/ There is also an existing deployment which will pull data from https://qa-reports.linaro.org/ and resides in this location (it is currently restricted to Linaro members):

```
https://datastudio.google.com/datasources/create?
↪connectorId=AKfycbxnkmVPXZRad22brXQ6BIB3iG9-GPWbjZnXds0vTuU
```

SQUAD Connector takes three arguments, token, group and project. The token argument is not required but then the dataset will be limited as for the non-authenticated user. After connecting it will display all the environments as metrics in the Data Studio, and it will use date and SQUAD metrics as dimensions. User can use this data to create reports and dashboards in the Google Data Studio as they see fit.

User is also free to deploy an instance of the Connector of their own using the code and manifest presented in the codebase.

CHAPTER

NINE

USE CASE: SETUP SQUAD WITH LAVA

9.1 Introduction

Once SQUAD installation is complete, a typical use case would be integrating it with a LAVA instance. The purpose of this use case is to describe a step-by-step set up of SQUAD with LAVA to submit and fetch jobs. If you haven't yet set up SQUAD, take a step back and follow *Installation Instructions for production environments*

9.2 Setting up a LAVA instance

LAVA has its own extensive documentation on how to get a server up and running. If you don't have it already, please refer to LAVA installation to configure yourself one. From this point it's taken that you have enough access to a running LAVA v2 instance that jobs can be submitted to and fetched from.

Note

Make sure that your LAVA instance has `event notifications` enabled, as it is disabled by default. See LAVA event notifications for details.

9.3 Creating a Backend for a LAVA instance

Log in into SQUAD admin view (`/admin`) and access `ci > backends > add backend` for inclusion form. Fill up accordingly:

- name: name of the backend. Example: validation.linaro.org
- url: LAVA RPC2 endpoint, it's how SQUAD will communicate with LAVA. Example: https://validation.linaro.org/RPC2
- username: a LAVA user with enough access to submit jobs
- token: a generated token for the user above, used to securely connect to the LAVA instance. See LAVA authentication tokens for details
- implementation type: leave it as `lava`
- backend settings: used to spare specific settings for LAVA instances. For details see *Backend settings*
- poll interval: number of minutes to wait before fetching a job from LAVA
- max fetch attempts: max number of times SQUAD will attempt to fetch a job from LAVA
- poll enabled: if this is disabled SQUAD will not try to poll jobs from LAVA

9.4 Creating a Project in SQUAD

SQUAD needs minimal data to start working with LAVA: Group and Project. By logging in the admin view, go to `core > groups > add` to add a new group and `core > projects > add` to add a new project. These are trivial to create, but please feel free to contact us if any help is needed.

9.5 Submitting and fetching test jobs

Given that all steps above are working correctly, you are ready to submit your first job through SQUAD. You can learn from LAVA documentation how to write new test definitions or use existing ones. For the sake of simplicity, we'll stick to LAVA's example of first job and use it to call SQUAD api for submitting a new test job:

```
wget https://validation.linaro.org/static/docs/v2/examples/test-jobs/qemu-pipeline-first-
↪job.yaml
curl localhost:8000/api/submitjob/<group-slug>/<project-slug>/<build-version>/<env> \
     --header "Authorization: token $SQUAD_TOKEN" \
     --form "backend=<backend-name>" \
     --form "definition=@qemu-pipeline-first-job.yaml"
```

Where `group-slug` and `project-slug` are the ones created in steps above, whereas `build-version` and `env` do not need to exist before submitting a job. For clarification `buid-version` is usually a git-commit hash and `env` is commonly the board target that a job is running. Although it's not covered in this tutorial, creating a `squad-token` is straightforward, do so by logging into admin view and go to `auth token > tokens > add` to add an auth token for a user. Lastly, `backend-name` is the one created in the sections above.

9.6 Extra use cases

The example above showed a simplistic SQUAD instance working along with a LAVA one. More can be done by using SQUAD's backend API to transform it into a proxy between a CI system (e.g. Jenkins) and a LAVA server. An instance of SQUAD is currently running at https://qa-reports.linaro.org and its set up is fully automated through ansible scripts at https://github.com/Linaro/qa-reports.linaro.org.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX