IBART Documentation

Linaro

Jun 21, 2023

Installation

The main documentation for the site is organized into a couple sections:

CHAPTER 1

Installation Process

1.1 Prerequisites

1.1.1 Ubuntu / Debian

Install the necessary packages using apt-get.

```
$ sudo apt update
$ sudo apt install git python3 python3-flask python3-pexpect python3-yaml python3-
↪requests
```

1.1.2 Arch Linux

Install the necessary packages using pacman.

```
$ sudo pacman -Syy
$ sudo pacman -S git python3 python-flask python-pexpect python-yaml python-requests
```

1.1.3 pip based

If you prefer working with pip based install instead of the above, then you should install

```
$ pip install --user pexpect pyyaml flask requests google-oauth2 \
                     google-api-python-client google-auth \
                     google-auth-oauthlib google-auth-httplib2
```

For Ubuntu and Debian based system you also need the pip package

```
$ sudo apt install git python3 python3-pip
```

Likewise for Arch Linux

```
$ sudo pacman -S git python3 python-pip
```

1.2 GitHub

1.2.1 Webhooks

First one needs to setup webhooks at GitHub. Important things to configure here is the `Payload URL`, which should point to the server running IBART. The listening port is by default `5000`. For `Content type` one should select `application/json`. The secret on the GitHub webhooks page is a string that you need to export in your shell before starting IBART (see "Running IBART/Exports"). At the section `Which events would you like to trigger this webhook?` it is sufficient to select `Pull requests`.

1.2.2 Personal access token

You need to generate a personal access token for your GitHub account. Note that this is not something unique for an individual git. You can generate the token at the Peronal access token page.

1.3 Google cloud console

This is a necessary step to be able to support authentication via Google login.

1.3.1 Create the app

First you need to create a web-application at the Google cloud console.

1.3.2 OAuth consent screen

Next step is fill in the OAuth consent screen. There you have to fill in the authorized domains and some support emails etc.

1.3.3 Create OAuth credentials

Last step is to create the OAuth credentials. Go to https://console.cloud.google.com/apis/credentials and press + *CREATE CREDENTIALS*, select *OAuth client ID*. For *Application type*, select *Web application* and give it a (any) name.

You need to add *Authorised JavaScript origins*, which typically is this for development https://localhost:5000 and for real domain it is something like https://mydomain.com:5000.

You also need to add *Authorised redirect URIs*, which is a list of URL that Google are willing to redirect you to after your Google identity has been authenticated. IBART needs the *callback* page to be enabled, i.e. for local development, this: https://localhost:5000/callback and for real use, something like this: https://mydomain.com:5000/callback.

Once completed, press save and you should find your new Oauth2 credential under OAuth 2.0 Client IDs. On the right side, press the arrow to download this credential. Save the file for now (there are "download" links/buttons to get the *json* file), we will use it later on (rename the file to *client_secret.json*).

1.4 Clone IBART

Obviously one need to clone IBART also, it doesn't matter where it is placed.

```
$ git clone https://github.com/jbech-linaro/ibart.git
```

CHAPTER 2

Running IBART

2.1 Exports

Note: Before starting IBART there are few **mandatory** exports that you need to do in the shell:

- `$ export IBART_URL=http://URL-to-the-server:5000`
- `$ export GITHUB_SECRET="The string you provided in the webhooks"`
- `$ export GITHUB_TOKEN="my-long-hex-string"` which you generated at the Peronal access token page.
- `$ export APP_SECRET_KEY="Some key that is hard to guess"` this signs the cookies (see Flask sessions for more information)

Besides that there are the following optional exports that will override what you have specified in configs/settings.yaml.

- `$ export IBART_CORE_LOG="/my/path/to/ibart/core.log"`
- `$ export IBART_DB_FILE="/my/location/to/ibart.db"`
- `$ export IBART_JOBDEFS="/my/folder/with/jobdefs"`
- `$ export IBART_LOG_DIR="/my/path/to/ibart/build-job/logs"`

2.2 Server Settings

Set up global settings in configs/settings.yaml. Note that quite a few of them are not in use in this current version.

2.3 OAuth file

The *client_secret.json* file downloaded using the install steps, needs to be placed in the root of the *ibart* git. However **don't** add it to the git tree, since that is a file containing sensitive data!

2.4 Default job definitions

Write a job definition and store it in `jobdefs/my-job.yaml`. Any name will do as long as it ends with `.yaml`. There are a few example jobs in the subfolder jobdefs/examples. That can be used either directly or as a template when writing your own. The script will ignore jobs in the examples folder, so either you have to copy the up one level or you have to symlink to them.

Since IBART runs all job definition it finds in the `jobdefs` folder in alphabetic order it is a good practice to prefix them with a number. I.e using symlinks one could do like this:

```
$ cd jobdefs
$ ln -s examples/optee_qemu.yaml 01-optee-qemu.yaml
$ ln -s examples/linux_kernel.yaml 02-linux-kernel.yaml
```

By doing so, IBART will first run the jobs `01-optee-qemu.yaml` and when that has completed it will continue with `02-linux-kernel.yaml`. You don't have to number like this, but the running order might not be what you expect if you don't do it.

At this moment it is only possible to use job definitions at the server. In the future we will add support for reading a `ibart.yaml` from the Git / pull request itself (see proof of concept branch https://github.com/jbech-linaro/ibart/tree/remote_yaml).

2.5 Starting IBART

Starting IBART is as simple as:

```
$ ./ibart.py
```

If everything done correctly, IBART should now be listening for build requests as well as serve HTML queries at `http://${IBART_URL}`

2.6 How jobs are processed

There are two ways to get jobs running. Either it comes as a webhook request from directly from GitHub or it is user request by a user to rebuild a certain job. For GitHub jobs the following happens:

- If it is a new pull request, then a new job will always be added to the queue.
- If it is an update to an existing pull request, then it will first cancel ongoing and remove pending jobs and then add the updated pull request to the queue. I.e., there can only be a single job in the queue for a given pull request when it is a build request coming from GitHub.

If it is an user initiated request (typically pressed `restart` or `stop`), then following applies:

- If a request affects a job already in the queue, then it will stop and remove it, then it will (re-)add the job to the queue.

CHAPTER 3

Job definitions - Yaml-files

This is the main thing a user will work with. This is where all commands to clone, build, flash etc takes place. There are `15` pre-defined sections and at this moment they are the only ones that can be there. You don't have to add nor use all of them. But you cannot add more or invent your own. A full file contains the following:

```
pre_clone:
clone:
post_clone:

pre_build:
build:
post_build:

pre_flash:
flash:
post_flash:

pre_boot:
boot:
post_boot:

pre_test:
test:
post_test:
```

3.1 Commands

Within each section one states commands, expected output and the timeout. Timeout (`timeout`) is by default `3` seconds if that is not stated. The expected output (`exp`) can be omitted if not needed. Most often one either writes a single command (`cmd`) or a combination with all three of them. Here is an example of how a job definition file could look like:

```
pre_clone:
    - cmd: mkdir -p /opt/myworking-dir
    - cmd: cd /opt/myworking-dir

clone:
    - cmd: git clone https://github.com/torvalds/linux.git
      timeout: 600

build:
    - cmd: make ARCH=arm defconfig
    - cmd: make -j8
      timeout: 600
```

This simple test would create a working directory, clone Linux kernel with a 600 second timeout, build it for Arm (again 600 seconds timeout). Note that one can use both this

```
:emphasize-lines: 3
build:
    - cmd: echo $?
      exp: '0'
```

as well as this syntax (pay attention to the added – at `exp`.

```
build:
    - cmd: echo $?
    - exp: '0'
```

From user point of view there is no difference. But under the hood, the later is done in two loops within the script and the first one is done in a single loop.

3.2 Exported variables

Under the hood IBART uses pexpect and for each section the job-definition file (yaml) it will spawn a new shell. This means that things are not normally carried over between sections in the job-definition file. But since it is both cumbersome and easy to forget export the same things over and over again, IBART saves every export it sees and when entering a new section it will export the same environment variables again. So, from a user perspective exports will work as expected.

3.2.1 Pull request variables

There are a few of the pull request variables automatically exported to the "environment" which can be used directly in the script, they are:

Variable	Meaning	Example
`PR_NUMBER`	The current pull request number	123
`PR_NAME`	The name git corresponding to the current pr number	ibart
`PR_FULL_NAME`	Both the GitHub project name and the name of the git	jbech-linaro/ibart
`PR_CLONE_URL`	URL to the submitters git/tree	https://github.com/jbech-linaro/ibart
`PR_BRANCH`	URL to the submitters branch	my_super_branch_with_fixes

3.3 Directory changes

Just as for the exported variables the last seen `cd` command is saved and then executed when spawning a new shell on for a new section in the job definition file. I.e., from user perspective a `cd` will carry over to the section in the job definition file.

CHAPTER 4

Security considerations

This is a very early version and there are things that are not secure:

- Anyone can restart and stop a job by going to the main page on IBART as long as they have authenticated themselves using Google. Anyone with a Google account can login and trigger rebuilds, cancel jobs etc. People abusing this will be banned.
- It runs Flask `debug` mode by default (consider using nginx for example instead of the Flask web server).
- Whatever is in the job definition file will be executed and it will do this with the same permissions as the server itself. So if one type `cmd: rm -rf $HOME` in the job definition file, then all files in the servers' $HOME folder **will** be deleted. So be very careful with what you or someone else puts into job definition file.