vlbuildbot Documentation

Release 1.0

M0E-lnx

February 06, 2017

Contents

Contents:

CHAPTER 1

Packaging guidelines for the VectorLinux buildbot

1.1 Introduction:

The packaging procedures for use with the VectorLinux Buildbot are a bit different, although the principles are still the same and some of the steps on the procedure remain unchanged. In the past, one would normally generate a build script (SlackBuild in most cases), then one would get the source code for the application as well as a fitting slack-desc file. When it was built, one would upload a complete source directory containing a tarball, description, and build script as well as the binary package itself (.txz as of VectorLinux 7.0).

With the buildbot system, things change a little bit. You still need to generate a script (See Buildscript Guidelines below), and you still need a description file. However, the only thing you upload is the build script and any supporting files it needs (slack-desc, patches, etc). See the guide for submitting a package below.

1.2 Why use a buildbot?

- Allows us to build packages for multiple architectures virtually parallel to each other. This means that the x86_64 version of every package in our repositories will be available at the same time the x86 package is ready, without the need for anyone to do any extra work.
- Makes package maintenance easier.
- Would allow us to release security fixes in a timely manner.
- Makes the job of the repo maintainer easier.
- Enforces consistency during the development of the distro by forcing developers and contributors to re-use the existing tools and resources as much as possible.
- Allows us to see which packages are failing and why so we can address the changes and get them fixed.

1.3 Submitting a new package

The following steps should be taken to submit a new package to the database.

1. Generate the build script with sbbuilder. (See Buildscript Guidelines below).
2. Get or write a slack-desc file for the application you are building. You can usually find them online. If they don't exist, you need to create one.
3. Execute the script to make sure it builds on your box. Mainly, this will make sure the LINK and build procedures are correct on the SlackBuild. Make any corrections to the SlackBuild as necessary.

4. Submit your SlackBuild, slack-desc and any other supporting files (patches, icons, .desktop files, etc) to the appropriate git repository for the vector release it should be built for. ie, for vector 7.1, upload your work to http://bitbucket.org/VLCore/vl71

1.4 SlackBuild Guidelines:

All SlackBuilds (with very few exceptions approved by a buildbot maintainer) submitted for inclusion on the buildbot system **must** meet the following requirements:

- Must be generated by sbbuilder. (some modifications for sbbuilder will be released soon for including stuff useful for the bot)
- Must include a value in the `LINK` array.
- Must list any and all dependencies required *to build the package* in an array named `MAKEDEPENDS`. This is an array separated by a single space character. See 'Handling Dependencies' below.
- The script must be sourceable. Meaning, we should be able to source the script from a shell without running it. This can be accomplished by adding a `if [ "$NORUN" != 1 ]; then` at any point after the BUILD ENVIRONMENT has been set in the script and the source is extracted. The `if` must be closed at the end of the script.
- Must be tested by the contributor to make sure it is free of syntax errors and to make sure it downloads its source package correctly.
- Must be accompanied by a slack-desc and any other files the SlackBuild needs to run successfully. This includes any patches, icons, .desktop files, etc.

All of these apply to every single submission. There will be special cases where an exception to the rules needs to be made. In such cases, communication between the package maintainer and the buildbot maintainer will determine how it will be implemented.

1.5 Editing existing (vectorlinux) scripts for compatibility with vlbuildbot

Existing vectorlinux build scripts can be adapted for compatibility with vlbuildbot. Most scripts only require 2 small adjustments.

- Fill in the `LINK` array.
- Add the `if [ "$NORUN" != 1 ]` condition.

DO NOT hard code package names and version numbers into `LINK`. The string in `LINK` must read something like: `http://somehost.com/$NAME/$NAME-$VERSION.tar.gz` instead of: `http://somehost.com/foo/foo-1.0.tar.gz`.

After this, simply test the script to make sure it does indeed download the source code, and submit

1.6 Handling Dependencies

The buildbot is equipped with a small tool to resolve any dependencies needed to build your package. But you must list them in the `MAKEDEPENDS` array of your SlackBuild. To solve these, the bot will first look in the official VectorLinux repositories. If a dependency is not in the repositories, it will look in the source repository for the release it's working

for to see if we have build scripts that will produce the package needed. If a dependency is not found at either location, your build will fail immediately.

Note: The `MAKEDEPENDS` array should only list dependencies needed **to build** your package, not to run it. Requiredbuilder will list all the dependencies needed to run the application you just packaged.

Note: Listing packages in `MAKEDEPENDS` forces the build slaves to always use the latest versions of the listed packages available.

1.6.1 How to list your build-time dependencies

Consider the following scenario.

You are building package 'foo'. But when you run the SlackBuild locally (for testing before submission), you find that the configure step needs 'libbar'. You would do the following.

- Edit your SlackBuild to add 'libbar' to your `MAKEDEPENDS`.
- Install libbar on your local system so you can continue to test the SlackBuild.
- Run the SlackBuild again.

Repeat those steps for every build dependency you need to list for your SlackBuild. If you need to depend on a package that is currently not in the official repositories, you will have to submit buildscripts for your dependency too.

Note: In rare cases, the package you are building may conflict with another package, or will not build if a certain package is installed. To have the build slave remove a package from the build environment before building your package, list your conflicting package with a "!" prefix (ie `"MAKEDEPENDS="!foobar"`).

Note: The default bahaviour for resolving build-time dependencies is as follows to first attempt to install the package from the repositories (via slapt-get). If that fails, then the package listed in `MAKEDEPENDS` will be built from source, and installed in the build environment before the requested package is built. **If the package fails to install from the online repos and from source code, the build for the requested package will fail until the dep is question is resolved.**

1.7 The life of a package

The following explains how a package is created and how it ends up in the repositories.

1. A contributor (packager) submits a buildscript to the git repository.
2. The buildbot master will detect the change on the git commit logs and instruct the build slaves to build the new package.
3. The build slave receives notification from the master that a new package needs to be built, and runs the script submitted by the contributor.
4. The build slave notifies the master of the build results. If the build was successful, the binary package and source code are uploaded to http://vlcore.vectorlinux.com/pkg/

5. Once the package makes it to vlcore.vectorlinux.com, it is picked up by other tools for processing and submission to the official VectorLinux repositories.

1.8 How the bot works

The VectorLinux buildbot is a special configuration of the buildbot tool found at http://buildbot.net It consists of at least 2 parts. One master, and at least one slave.

1.8.1 The Master

The buildbot master is responsible for the following tasks:

- Monitor the git source repository for changes.
- Assign tasks for each build slave as necessary.
- Collect results data and build logs from the build slaves.
- Expose build results and logs via a web ui.
- Provide an IRC bot as another way to force builds and notify the dev channel of events taking place.
- (currently disabled) Provide email notification of triggered events.

1.8.2 The Slave

The buildbot slave(s) are responsible for the following tasks:

- Receive instructions from the build master.
- Carry out the actual build process when instructed by the master.
- Relay build results and log files back to the master.
- Upload the resulting binary package and source tree to the pool location.

CHAPTER 2

Package maintenance guildelines

This guide explains how package maintenance is done for packages contributed to the VectorLinux Buildbot system. The following guidelines apply to all package maintenance performed to every package in the database.

2.1 General Guidelines

- All package revisions **Must** increase the `build` value on the updated package. This will help `slapt-get` detect the available update on all end-user's machines.
- All version bumps (increases) **MUST** reset the `BUILD` value back to 1
- All SlackBuilds must be tested by the contributor before they are submitted.

2.1.1 How to maintain a package

When performing maintenance on a package, always follow these guidelines to make sure your contributions are in line with the rest of the development efforts.

- Change to the directory holding the clone of your repository. This is the location where you ran `git clone http://bitbucket.org/VLCore/vlxx`. The slackbuilds can be found in the `var/vabs` directory of each reposiroty.
- Locate the application(s) you want to maintain or update, and open the SlackBuild with your favorite text editor.
- Make your changes to the build script and save it (with the same name). Changes should include any changes in package version, build number, URL to source tarball **direct download**, or other necessary build procedure changes.
- Run the script to test it's basic functionality.
- When you have verified that the script runs and does not return an error, then you commit your changes, and push them to the online git repository.

Note: **ALWAYS** run `git pull` before you decide to modify any SlackBuild and before you run `git commit`. This will avoid conflicts and makes for a good workflow betweek many contributors at the same time for the same repository.

2.1.2 Typical maintenance example

A typical package maintenance procedure would be necessary when a new version of an application is released from upstream. At that time, the packager would need to do the following.

- `git clone https://bitbucket.org/VLCore/vl71.git`
- `cd vl71/var/vabs/`
- Find the SlackBuild in the repository that builds the application that was just released.
- Edit the SlackBuild to update its `VERSION` value
- Reset the `BUILD` value back to 1 (with every version bump, the release should be reset back to 1)
- Make sure the `LINK` value can download the new version of the source tarball.
- `git pull origin master`
- `git add your_modified.SlackBuild`
- `git commit -m 'your commit message goes here'`
- `git push origin master`

You are done.

2.2 FAQ

Q: If I have to test the SlackBuild myself, why can't I just upload package instead of the SlackBuild?

A: Contributing a SlackBuild results in multiple packages for different architectures created from your SlackBuild. Uploading a package only contributes to **your** current system.

CHAPTER 3

Package Testing

This guide explains how to use the built-in package testing facilities inside the vlbuildbot build system.

3.1 How it works

The vlbuildbot system offers the ability to perform tests on the resulting package(s) created by the `.SlackBuild` file. The testing system is pretty much wide open for the contributor to write their own tests for the package they are contributing. In a nutshell, this is how the process goes.

- The `.SlackBuild` file is executed. If this exits in error, the build is marked as failed, and everything halts there.
- After the `.SlackBuild` exits, the bot will look for a file named `<package_name>.tests.sh`, where `<package_name>` refers to the name of the program you are building. For example, the `htop/src` directory will contain `htop.SlackBuild` and `htop.tests.sh` if testing is desired. If the `.tests.sh` file is found, the bot will install the package it just created and execute the `<package_name>.tests.sh` file. If this program returns an error, the build is marked as failed, and the package is discarded.

3.2 How to write tests

The syntax to a `tests.sh` file is just plain shell script. You can assume the resulting package has been installed by the time this script is executed.

The idea behind te tests file is to verify that a package works as it should, so you could run any number of tests against the package. For instance

- Check the existance|permissions|version of a file after the package is installed.
- Use the installed program to perform a simple task to make sure it works as it should.

3.2.1 Sample use case

When testing packages like `python`, `perl`, `lua` or any other programming language package for example, you could also place a sample program written in that language and execute it from your `tests.sh` file.

3.3 Requirements on the tests script

Generally, the tests script is wide open to whatever you want to test for using shell syntax. In order to properly work though, the following criteria must be met by the `.tests.sh` program.

1. The tests program must follow the same naming convention used throughout the vlbuildbot system. For instance, the `htop` program will have a `htop.SlackBuild` and a `htop.tests.sh` file is testing is desired after the build completes.
2. **The script must return a value (ie, if it errors out, return a value greater than 1).** Not having this **will produce false positives and the render testing useless**
3. The script must be placed in the same directory as the `.SlackBuild`
4. All resources required by the `.tests.sh` must be provided and placed

   on the same location as the tests file, ie, a test `hello world` program to be compiled as a test.

Note: The tests are performed in the same environment that was left after the package was built. All packages installed as `MAKEDEPENDS` of the `.SlackBuild` are still present when the tests run. Any run-time dependencies your package may need, and that are not present on the default building environment can be installed via `slapt-get` from the `tests.sh` program.

3.4 Known Limitations

As of the time of this writing, the package testing has the following known limitations.

1. X applications cannot be launched from the `.tests.sh` file. This is because the X Window System is not reachable by the docker container where the tests are ran. This limits the ability for instance to launch a GUI application and take a screenshot.

Note: Due to the fact that this is the first attempt at automatic package testing, more limitations or caveats to this rough-draft implementation may surface. In a case like that, a bug report should be filed at http://bitbucket.org/VLCore/vlbuildbot

---

CHAPTER 4

Repository maintenance guildelines

This guide explains how the maintenance of the official VectorLinux repositories is done when working with packages contributed to the VectorLinux Buildbot system.

At the present time, there are no tools to automatically move the packages to the official repositories. When a packager contributes a package or package update, the buildbot will build as instructed, and place the resulting packages in http://vlcore.vectorlinux/pkg/untested.

The repository maintainer will have to move the packages like they usually do now, with the exception that each package should be tested before moving it to the repositories to make sure the package works as expected.

4.1 What you need to know

- The bot collects packages in http://vlcore.vectorlinux.com/pkg/untested
- Source code for the packages can can be found at http://vlcore.vectorlinux.com/src/
- You should delete the packages from the untested repository once the package has been moved to the official repositories.
- You must trigger the scripts to update the metadata on the untested repository as well as the official repositories when you perform maintenance.

CHAPTER 5

Repository Guidelines

5.1 Repository Layout

Like all other distros, the VectorLinux repositories are divided into different areas. Traditionally, these are named as follows.

5.1.1 packages

This area of the repositories holds the packages that ship on the ISO image at the time of the release. These will be outdated as development progresses, but are to be kept there for long term support. These packages will remain there and will not be updated.

Note: In order to maintain stability and support, the packages in this section do not receive further updates.

5.1.2 patches

This area contains stable, tested packages that represent updates and and security fixes to the packages in the `packages` section. The packages should be moved to this area only after being reasonably tested.

5.1.3 extra

This area contains packages that are considered stable enough but that are not vital to the system. These packages have been used to fulfill dependencies on the vlbuildbot package building process. This area will contain other featured packages that are not vital for a working system installation but offer addditional functionality on top of the base system. These packages are well supported but do not necesarily ship with the ISO image. The packages in this area will receive updates.

5.1.4 untested

This area will contain the most current bleeding-edge versions of packages. The vlbuildbot system will upload packages to this area directly, so that means the packages in this area are the most unstable used for development and testing purposes. Packages in this section are automatically purged every week.

5.2 Preparing for Distro Release

While the distro is preparing a new release, all current packages will remain in the untested area. This allows for testing ISO images to be built and tested as a whole.

Once the testing ISO is stable enough to release the final product, the following steps should be taken to create the stable areas of the repositories for the new release.

1. All packages that make up the BB (buildbase) ISO image should be moved to the `packages` section of the new stable repository. These will not receive any kind of updates at all.
2. All other packages that are in the untested repository but wont ship on the final ISO image should be moved to the `extra` section of the repository.
3. The `patches` section of the repositories will start off empty, most likely with empty metadata just so slapt-get wont fail for the users.
4. The `untested` repository will enter automatic purging mode limiting the life of untested packages to one week from the time it gets built and uploaded by vlbuildbot. That two week period should be enough for stabilized packages to make it to `extra` or `patches` as needed.
5. Development will begin on a new release, so a new untested repository will be needed for the new release.

Note: The automatic purging cycle does not affect the `untested` area for a release while in development.

Note: Packages that exist in the `packages` section should not exist in the `extra` section. If a package from `packages` requires a security or bug fix, that update goes in the `patches` section of the repositories.

CHAPTER 6

REPOSITORY MAINTENANCE TOOL SPECS

6.1 Introduction & Project Goals

This document outlines the spec of work for a tool required by the Vectorlinux project to manage package repositories.

The implementation of an automated packaging system has allowed great progress on the overall distro, but at the same time has raised some issues regarding the package repository structure and maintenance methods. The goal of this document is to provide the specs for a tool (to be named later) that will allow the distro developers to automate or ease the task of organizing the binary repositories and streamlining the movement of packages around from one section of the repositories to another.

6.2 Proposed Workflow

All packages built by the automated build system are pushed to the vlcore server. This tool would allow a package maintainer to easily take these packages allocate them to the correct location.

The repository maintainer will test the package directly from the untested repository to make sure it works as expected and will either approve or reject the package. Approved packages will be re-located to the appropriate section of the repositories.

After a package has been relocated to the appropriate section on the repositories, the copy from the vlcore server (untested repository) should be removed.

6.3 Implementation

The tool should consist of a web-based application that would list the packages recently uploaded and offer the user the option to approve or reject the package.

If the package is approved, it will be allocated to the appropriate section of the stable repositories. Otherwise, the package will be deleted and a bug report must be filed on the SlackBuild that built the package indicating the problems with the package and the reason the package got rejected.

Packages will be “approved” or “rejected” by managing metadata or “tags” Sample tags for approved packages can be either `EXTRA` or `PATCHES`. After that more tags can be applied to indicate which section of the repository it should go in. These additional tags could be the name of the sub-directory inside the repository section. Rejected packages should be tagged as `REJECTED` and the package will be discarded immediately.

After every maintenance session, the slapt-get metadata should be updated to propagate changes to all end users.

Note: The web-based application should be available to authorized users (repository maintainers) only

CHAPTER 7

Vlbuildbot (bot) maintenance

7.1 Maintenance on the bot master.

The buildbot master needs to be restarted when at least one of the following events take place.

- Adding or removing an application from the manifest.
- Adding or removing a slave from the bot system.
- Changes to the git polling frequency intervals.
- Changes in the master's master.cfg configuration

Changes are only applied when the bot master is restarted.

7.1.1 Adding a new slave to the master.

To add a new slave on the vlbuildbot system, the bot administrator needs to follow these guidelines:

Edit `slavenodes` on the master's home directory following the existing syntax. Each line represents one slave and is setup to contain four fields. `node_name | password | build_type | build_capacity`.

- `node_name` represents the name of the slave.
- `password` sets the password the slave will use to authorize with the master
- `build_type` is one of "pkg" or "iso". Slaves set to "pkg" will build packages while others set to "iso" will build ISO images from packages.
- `build_capacity` represents the number of simultaneous builds this slave can run at the same time.

Note: The buildbot's home directory is the directory containing the master.cfg file

7.2 Maintenance on the slave.

The slaves almost never need to be maintained, however, the status of each of the attached slaves can be seen at http://vlcore.vectorlinux.com/buildbot/buildslaves

That page presents a table 7 columns. The column to the far right indicates the current status for each slave. You may need to scroll down the page to see the status because we have a lot of builders listed. If the status says `offline` for any of the slaves, usually a slave restart will take care of this.

7.2.1 Updating the build environments on the slaves.

Buildslaves are setup to automatically check the `VLCore/vlbuildbot` git tree for available updates. Just about any update can be performed by pushing the necessary instructions to the git tree.

The `updated` directory on that git tree contains a file named `update.conf`. This file is used to define in simple `BASH` script how the update is rolled out.

Only (2) parts of the file **MUST** be changed for an update to be executed.

1. `REMOTEVERSION`. This is normally set to a date value (ie 20130423). The update system compares this value to the existing value in the build environment and determines if an update is needed.
2. `function update_slave()`. This bash function contains the instructions that make up the update. Wether the update consists of renaming a file, or updating the entire build environment with new ISO's, it's all done here.

Rolling out updates this way helps a lot with maintenance on the buildbot system for VectorLinux. Since every slave is watching this git repository, all slaves will be updated simultaneously.

Note: If a slave is busy at the time the update is detected, the slave will request a graceful shutdown from the master. This means the slave will shut itself down when the running build finishes (pass or fail). Then the bot will update itself and come back online.

Note: Build slaves are setup to check for updates every hour, so not all slaves will be updated at the exact same time.

7.3 Preparing a new slave.

Having multiple slaves attached to the buildbot master allows for simultaneous maintenance of all tracked packages and spreading the workload between all the active buildslaves. When more than one slave is suited to build the requested package, the buildbot will pick one at random to build. The following steps can be followed to prepare a new slave and get it ready for connecting to the buildbot master.

7.3.1 Minimun Requirements (on the slave host)

To host a slave instance, you will need 64-bit hardware and at the very least meet the following requiements.

- Hardware
- 1024M RAM.
- 30GB available disk space.
- 2GB swap space.
- Software (must be installed and tested to work inside the isolated environment)
- procmail
- git
- dev-base (metapackage)
- kernel-headers
- setuptools

- lftp
- wget

7.3.2 Preparing the slave host

Build slaves normally run in an isolated environment. This can be either a virtual machine or a LXC container. This guide will not cover how to set that up, but will assume the procedure is performed within an isolated environment. For a quick guide on LXC containers on VectorLinux or VLocity, see http://vlcoredocumentation.readthedocs.org/en/latest/manuals/lxc-containers.html

The `deployment` directory of the `VLCore/vlbuildbot` git tree contains the necessary tools to deploy a new slave. The following procedure must be carried out to deploy a new slave

1. `git clone http://bitbucket.org/VLCore/vlbuildbot`
2. `cd vlbuildbot/deployment`
3. Edit `mkslave.sh` and fill in the following fields:
   - `SLAVE_NAME`: Should be the node name on the master for this slave
   - `SLAVE_PASSWORD`: Password used to authorize this slave at the master.
   - `MASTER_HOST`: IP or URL to the bot master
   - `MASTER_SLAVE_PORT`: Port on which the master expects slave connections.
4. `cp ../slave/etc/vlbuildslave/slavehost.conf .`
5. Edit `slavehost.conf` and set the following fields as follows:
   - `SLAVES_ROOT`: Path to where the slave will live.
   - `JAILS`: Path to directory where the jails will be kept.

     Note: This directory will require lots of disk space. This is where the read-only areas are kept for clean builds. This could require up to 20GB of disk space (subject to change)

   - `REPOS_HOME`: This is where the git clones of the SlackBuild repositories will be kept.
   - `TOOLSDIR`: This should always be set to `/usr/local/bin`

7.3.3 Launching the slave deployment process

After all the above steps have been taken, you will need to launch the deployment process from the same directory your mkslave.sh is at.

1. Make sure the correct timezone has been set on your host. This can be checked

   by making sure `/etc/localtime` and `/etc/localtime-copied-from` point to a valid symlink.
   If they do not, fix it before you start the deployment process.

1. `export VLBB_REPO=$PWD/..`
2. `export CMD_MKCHROOT=$VLBB_REPO/slave/sbin/mkchrootSB`
3. `export SLAVECONF=$PWD/slavehost.conf`
4. `sh mkslave.sh`

7.3.4 After deployment

After the slave has been deployed, you will need to do a couple of things before the slave can be used by the master.

1. `source /etc/vlbuildslave/slavehost.conf`
2. `cd $REPOS_HOME`
3. `git clone http://bitbucket.org/VLCore/vl70`
4. `git clone http://bitbucket.org/VLCore/vl71`
5. `git clone http://bitbucket.org/VLCore/vl72`
6. You should also provide the bot master's administrator with a set of your root's ssh public keys. This is needed to allow your slave to upload built packages.

7.3.5 Test your setup

You can test your new slave by manually launching a build as follows.

`/usr/local/bin/vlbb-jailedbuild -p htop -i 1 -a i586 -v veclinux-7.1`

This will trigger a local build of htop for 7.1 32-bit on your isolated environment. Repeat the step by issuing the command and changing the `i586` to `x86_64` and the `veclinux-7.1` to `veclinux-7.0` or `veclinux-7.2`.

As a rule of thumb, a slave should be tested with the above procedure before connecting it to the master. The slave should be tested to build a package for every release for every architecture supported. If any one build process fails, the slave is not working and cannot be connected to the master.

When testing a slave, use a simple package to build. Htop is a good package to use because it does not take much to build. Normally, if a slave cannot build htop, it will not work at all.

If all tests pass, then the slave is ready to be connected to the master and to begin receiving build tasks.

7.3.6 Starting and Stopping the slave

After your slave has been setup and tested to be working, and you have been given the assigned node name and password for your slave, you may start the slave by using the provided rc script

`/etc/rc.d/rc.vlbuildslave [ start | stop ]` will start or stop the slave.

Note: The deployment process will begin by first downloading all of the necessary ISO images to create the required jails. This will take time, bandwidth and disk space.

Note: In the VectorLinux buildbot system, all buildslaves are *required* to be able to build for both the x86 and x86_64 architectures. This can be easily acheived by setting up the 32 and 64b chroots properly. Setting the slave up in a virtual machine of any kind allows more flexibility and independence from the real hardware.

CHAPTER 8

Interacting with the VectorLinux Buildbot

User interaction with the VectorLinux buildbot is mostly automated. The bot monitors the the source repository configured at the master. Any activity detected by the master will automatically trigger the bot to react accordingly. Any change on any application included in the manifest at the master will trigger that application to build.

8.1 Working with git

To many of us, git can be intimidating at the beginning, but you really dont need to know it inside out to be able to work with the bot. Here is what you do need to know.

Note: When working with vabs git tree, **avoid using the git gui**. Use a terminal application instead. Get used to it. You'll get a better understanding of how git and the vabs git tree work that way.

8.1.1 About the vabs git trees

Our buildbot is configured to monitor several VectorLinux releases as of this writing. Each release in represented with a separate git repository under the VLCore project at bitbucket.org as follows

- Vectorlinux 7.0: http://bitbucket.org/VLCore/vl70
- Vectorlinux 7.1: http://bitbucket.org/VLCore/vl71
- Vectorlinux 7.2: http://bitbucket.org/VLCore/vl72

The contributor should clone the appropriate repository for the vector release they are trying to build the packages for.

8.1.2 Example (Submitting builds for vectorlinux 7.0)

This is a short outline of how to submit builds intended for vectorlinux 7.0. This short howto assumes you are working with a terminal application.

- **`git clone https://bitbucket.org/VLCore/vl70.git`** This command downloads a copy of the remote vabs tree to your local directory.
- Make your changes to the SlackBuild you want to have built by the bot.
- **`git add <name_of_your.SlackBuild>`** This will add your change to the list of changes to be published.

- **`git commit -m "Your summary of your changes here"`** This will save your changes to your local copy of the git tree. When commiting changes, use a descriptive commit message. This lets other contributors know what you by just glancing over the commit logs.
- **`git push origin master`** This will publish your changes to remote git tree and trigger the build on the build slaves.

Note: The git push command **MUST** include the name of the branch you are working with. The buildbot will only monitor changes to the `master` branch of each repository. Changes to all other branches are ignored.

Note: If you are working with git over https (exactly as described here), the git push command will ask for your use name and password. If you checked out the git tree using ssh (requires ssh keys @ bitbucket), you will not be asked for credentials.

Note: The procedure outlined above works for all monitored branches.

8.1.3 Forcing the bot to ignore your change

By default, the vlbuildbot system is monitoring every change made to the git tree, and reacts by executing the changed scripts. To have the bot igore a specific change, you can add the keyword `!:nobuild` anywhere on your commit message.

```
git commit -m 'This build will fail until deps are fixed !:nobuild
```

8.2 Interaction via the IRC bot.

It is also possible to interact with the bot via the IRCbot plugin. The IRC status plugin is currently available at the #vectorlinux-pkgs channel on freenode. The bot can take commands to force builds, and to check on any specific builders.

Note: The IRC plugin for the buildbot is present on #vectorlinux-pkgs with nick `vlbuildbot` as of this writing.

8.2.1 IRC Bot Commands.

Here are some useful commands you can try with the IRC bot. All commands should be issued in the the following format. `botname: <command>` or `botname, <command>`

- `vlbuildbot: commands` This will list all the available commands for the bot.
- `vlbuildbot: help <command>` Displays additional help on <command>.
- `vlbuildbot: force build --vlrelease=veclinux-x.x <application> <reason_for_forcing_build>` Trigger an un-scheduled build of <application>.

8.2.2 Build time dependencies (MAKEDEPENDS)

By default, the buildslave will fulfill the necesary dependencies to build the requested package as instructed in the `MAKEDEPENDS` variable of the build script. The default behaviour is to install these dependencies from binary packages in the existing repositories. If that fails, then it will build the required package (that failed to install from slapt-get) from source before continuing with the requested package. There is a special provision for cases where you may need to compile a group of packages from source (ie, when updating one package breaks others (ie, ffmpeg)). The way to do that is by listing all the packages in question in the `MAKEDEPENDS` list, and forcing the build via the IRC bot with a special argument as follows.

```
vlbuildbot:  force build --vlrelease=veclinux-x.x --props=build-deps=TRUE
<app> reason for build
```

Note: The application name (ie, 'htop') will trigger a new build of the package for both architectures.

Note: The build-deps property can only be applied to builds via the IRC bot or via an authorized session at the bots web interface.

8.2.3 Special Build Properties Provisions

The following build properties can be specified when forcing a build from the IRC interface.

- **`build-deps=YES|TRUE|NO`** Makes the slave build the specified MAKEDEPENDS from source code.
- **`metabuild=YES|TRUE|NO`** Sets the METABUILD environment variable in the build environment. This can then be read in the SlackBuild to help with the creation of metapackages.

To set these environment variables in the build slave, add the following syntax to the force build command:

```
--props=build-deps=YES metabuild=YES
```

8.3 Interaction via the web interface.

The web UI is not just informational. It also provides a way to force builds of specific builders.

- Click on Builders.
- Find the application you want to force and click it's name.
- Fill in the `reason` field on the page.
- Click the `Force Build` button.

Note: The Web UI is setup to *NOT ALLOW* forced builds by default. If you need to be able to trigger builds from the Web ui, contact the bot master administrator to get login credentials. The `Force Build` button may be hidden until you log in to the web ui with the credentials given by the bot master administrator.

There is also a `Ping Builder` button which will force a 'refresh' of the available Buildslaves for this particular builder.

CHAPTER 9

FAQ (by packagers)

1. I need to add (one or more) applications to the bot. How do I do that?

   There are a couple of ways to do this.

   (a) Via the IRC channel by issuing the command `addaplication foo[,bar,foobar]`.

   (b) Submit a list of the applications you want added as a bug report to the vabs project at http://bitbucket.org/VLCore/vabs

2. The application i'm building needs `foo` installed, otherwise it will tail.

   Add a `MAKEDEPENDS` line on your SlackBuild and list your build time deps there.

3. The application I updated breaks some other package. I need to revert to an older version

   Update the SlackBuild again to make it build the older version (known to work). The broken version will be replaced by the older (working) version.

CHAPTER 10

PACKAGER TIPS

10.1 How to avoid failed builds.

The buildbot will automatically detect changes pushed to the SlackBuilds in the git tree. However, an application may require a new dependency on a newer version that it was not necessary to build older versions. This will result in a failed build. To avoid failed builds, test your SlackBuild on a clean build environment before you submit your change to the git tree. This can be accomplished by keeping a build JAIL on any vectorlinux system (7.0 and newer).

10.1.1 How to setup a clean testing environment (sandbox)

Use this procedure to replicate the same environment as what is used in the buildbot slaves to build your packages.

The following procedure should be ran as root user on an environment that has a configured working slave setup. (you should have a /etc/vlbuildslave/slavehost.conf)

For this example, we will assume you will work with for the Vectorlinux-7.1 environment in 32-bit. You will need to locate the path to the CHROOT-RO jail for this environment. This can be found by sourcing `/etc/vlbuildslave/slavehost.conf` and looking at the VL-7.1-BB-XN.N/ directory.

1. `source /etc/vlbuildslave/slavehost.conf`
2. `cd /root`
3. `mkdir devel`
4. `cd devel`
5. `mkdir rw`
6. `git clone http://bitbucket.org/VLCore/vl71`
7. `cd rw`
8. `REPO=/root/devel/vl71 ROPATH=${CHROOTS\[32c7p1\]} /sbin/sandbox`

This will give you a shell prompt with an AUFS read-only layer from the same environment the buildslave uses to build packages assigned by the master. Your git repository is mounted at `/home/slackbuilds` inside this shell. All changes made to your /home/slackbuilds directory are saved even when you exit the jail. When you are done working your tests, you may issue `exit` to return to your normal shell

---

Note: Packages installed inthe chroot environment **will not affect** the chroot when used by the slave. They are only saved in the 'rw' directory as the read-write layer. This layer can be removed and if you issue the sandbox command again, the jail will be reset to the default settings with the default set of packages reverted.

- search