Jabberwocky User Manual

Ian Orzel
Dylan McDougall

April 2023

Contents

Chapter 1

Tool Overview

This chapter provides a brief overview of the tool while avoiding much of the detail that is described in the later chapters. This can be used for anyone who is interested in learning about the basics about what this tool is and what it is for.

In section 1.1 we describe the motivations that led us to create this tool. In section 1.2, we describe a list of the major features that the tool fulfils. In section 1.3, we define what a container is for future reference. In section 1.4, we describe what the container manager is and what it does. In section 1.5, we explain how to install this tool onto your computer.

1.1 Motivations

Some courses require specific environments for completing your coursework. For instance, in the Compiler Theory course, students are tasked with creating a compiler for the SPARC architecture. The roadblock for this is that SPARC is an old architecture which modern computers do not support or utilize. Thus, to construct the compiler for the course, students must be able to run binaries on an architecture other than their own.

For these courses, students typically are tasked with getting them set up and working on their own computer. This can take up time for the students as well as time for instructors, as instructors often need to help them through this process. Beyond taking time, it can create situations where coursework works properly for the student, but not when the instructor runs it. This can cause students to get an unfair grade on their coursework.

The Jabberwocky tool hopes to solve these problems. Jabberwocky allows instructors to create containers containing specific virtual environments that students can easily install and use. Then, because students and faculty have identical environments, discrepancies between students and professors are less likely to happen.

1.2 Core Features

- Use on-demand virtual development environments custom-tailored to your courses. These virtual environments contain all the necessary custom software you need to develop programs for assignments, as well as to test and debug your software, all wrapped up in a tidy container that you install once.
- Use the pre-made environments created for the Compiler Theory, Programming Languages, and Operating Systems courses.
- Easily emulate foreign guest architectures other than that of your host machine.
- Run these development environments on all the major platforms: Windows, macOS, and Linux.
- Transfer files between your host's native filesystem and the container's virtual filesystem.
- Access and interact with the virtual shell of the container.
- Be able to create containers with custom software and distribute them to your students for a specific course.

1.3 Containers

From an abstract level, a container represents a virtual environment containing all of the needed tools to complete particular tasks. These tools can include architecture, software, and files. These containers are light-weight virtual machines that emulate a computer on your machine. These virtual machines are created using QEMU.

1.4 The Container Manager

The container manager is a piece of software that allows users to interact with containers installed on their computer. Using this manager, students can:

- Install a container
- Start a container
- Enter the container's shell
- Run a command in the container
- Send a file to the container
- Get a file from the container

- Stop a container
- Uninstall a container

The container manager can be used to manage multiple containers simultaneously. It is highly recommended that the container manager is used to interact with containers instead of trying to use QEMU directly.

1.5 Installation Guide

Installation of the tool is easy. The GitHub releases page contains installers for Windows, MAC (both Intel and M1), and Linux (x86_64). Once obtained, running this installer should fully install the tool. You may run into issues with the QEMU installation, in which case you will need to install QEMU manually. This should not be difficult, and there are many tutorials online about how to do this.

1.5.1 Build from Source

If you run into issues with the installer, you may choose to build the tool from source. This will take a bit more work, but it will arrive at the same result.

To start building, first ensure that you have Python 3.12 installed. Then, you will want to install poetry through pip. Here is an example of doing this:

```
$ apt-get install python3
$ python3 -m pip install poetry
```

Listing 1.1: Installing Poetry

This installation may be slightly different depending on the platform you are using. If you run into issues, there are many resources online for installing poetry on your system.

After that, you can build from source using the following commands:

```
$ git clone https://github.com/Kippiii/jabberwocky-container-
    manager
$ cd jabberwocky-container-manager
$ poetry install
$ poetry shell
$ python build.py
$ ./build/dist/installer-[platform]-[architecture]
```

Listing 1.2: Building from Source

Please note that platform and architecture should be filled in with the proper information (note that there should only be one file in the build/dist directory if you unsure about your system's information).

Chapter 2

For Students

This chapter provides the details that students need to know in order to use this tool to complete their coursework. It is also highly recommended that instructors read this section so that they can help students with the tools as well as simulate a student environment themselves.

In section 2.1, we define the concept of an archive and explain why it is used. In section 2.2, we provide a brief description of every command that a student may need (we do not recommend reading through all of these and instead recommend reading them as needed). In Section 2.3, we provide some examples of common scenarios that students will likely go through.

2.1 Archives

An archive is a file that contains all of the information to create a container on your local machine. Thus, given an archive, you can install the corresponding container onto your system. Similarly, you can convert your containers to archives in order to send them to other computers.

Non-abstractly, a container is a tarred archive that consists of two files: a QEMU hard-disk and a config json file. The QEMU hard-disk is a representation of the virtual machine that QEMU runs (thus, you could theoretically run this hard-disk without using our tool, but this is not recommended), and the config file contains information about how to start and manage the hard-disk.

2.2 Container Manager Commands

For this section, we will provide a run-down of every command available on the container manager. This list is expansive, and it is likely that the typical student will not use a majority of them. Let this act as a reference when unsure of how to use a command. For information on common usage situations, please see Section 2.3.

2.2.1 Help

```
$ jab help
```

Listing 2.1: Help command

Prints a help menu that contains information for all of the commands that can be used with the tool.

2.2.2 Install

```
$ jab install [archive_path] [container_name]
```

Listing 2.2: Install command

Installs a new container to your system from an archive. The archive_path parameter specified the path to the archive file on your system (can be relative or absolute). The container_name is the name you want the container that you install to have.

2.2.3 Delete

```
$ jab delete [container_name]
```

Listing 2.3: Delete command

Deletes a container from your manager (so it can no longer be started). The container_name is the name of the container that you are deleting.

2.2.4 Rename

```
$ jab rename [old_container_name] [new_container_name]
```

Listing 2.4: Rename command

Renames a container within your manager. The old_container_name is the current name of the container. The new_container_name is what the name of the container will be changed to.

2.2.5 List

```
$ jab list
```

Listing 2.5: List command

Provides a list of all containers that are installed on your computer.

2.2.6 Archive

```
$ jab archive [container_name] {archive_path}
```

Listing 2.6: Archive command

Exports a container to an archive on your system. The container_name is the name of the container that you are archiving. The archive_path is the path that you want the archive to be saved to. If this path is not specified, it will be saved to the current directory. If the given path is a directory, the archive will be saved with the file name of [container_name].tar.gz.

2.2.7 Start

```
$ jab start [container_name]
```

Listing 2.7: Start command

Starts the virtual environment of a given container, allowing the user to interact with it. The container_name is the name of the container that you are starting.

2.2.8 Stop

```
$ jab stop [container_name]
```

Listing 2.8: Stop command

Stops a container that is running on your system. If you do not stop a container, the container will continue to use resources on your computer in the background. It is highly recommended that you stop a container when you are done using it. This command also ensures that a container is stopped gracefully, and if your container is not stopped gracefully, it can cause future issues with the container. The container_name is the name of the container that you are stopping.

2.2.9 Kill

```
$ jab kill [container_name]
```

Listing 2.9: Kill command

Ungracefully stops a container. WARNING: Only use this command if the stop command fails. This command could cause future issues within the container that it is used on. The container_name is the name of the container that you are stopping.

2.2.10 Interact

```
$ jab interact [container_name]
```

Listing 2.10: Interact command

Opens the shell of a given container. Keep in mind that a container is really just a Debian virtual machine running in the background, so this command will ssh you into its shell. This allows you to run any number of commands in the container, get the output of the commands, and provide them with standard input. The container_name is the name of the container whose shell you are accessing.

2.2.11 Run

```
$ jab run [container_name] [cmd]
```

Listing 2.11: Run command

Runs an arbitrary command on the container, linking your standard inputs, outputs, and errors to that of the container. WARNING: It is highly recommended to use the interact command instead of this one. This command was created to be used with scripting, rather than by a user. The container_name is the name of the container that you are running a command on. The cmd is the command to be run in the container.

2.2.12 Send File

```
$ jab send-file [container_name] [src_path] {dest_path}
```

Listing 2.12: Send file command

Sends a file from your local file system to the file system on the container. The container_name is the name of the container that you are sending a file to. The src_path is the path to a file on your local file system that will be sent to the container. It cannot be a directory. The dest_path is the path on the container's file system where the file will be saved. If it is not specified, the file will be saved to the root directory. If a directory is specified, it will save to that directory with the same file name as on your system.

2.2.13 Get File

```
$ jab get-file [container_name] [src_path] {dest_path}
```

Listing 2.13: Get File command

Gets a file from the container's file system and saves it to your local file system. The container_name is the name of the container that you are getting the file from. The src_path is the path to a file on the container's file system that will be sent to your local file system. It cannot be a directory. The dest_path is the path on your local file system where the file will be saved. If it is not specified, the file will be saved to the current directory. If a directory is specified, it will save to that directory with the same file name as in the container.

2.2.14 Files

```
$ jab files [container_name]
```

Listing 2.14: Files command

Opens a light-weight FileZilla session that is connected to the container. This provides users with a graphical user interface for transfering files between their local file system and the file system of a container. The container_name is the name of the container whose file system you will interact with.

2.2.15 SFTP

```
$ jab sftp [container_name]
```

Listing 2.15: SFTP command

Opens an SFTP session with the container, allowing file transfers between the local file system and the container's file system. The container_name is the name of the container whose file system you will interact with.

2.2.16 SSH Address

```
$ jab ssh-address [container_name]
```

Listing 2.16: SSH Address command

Returns the information needed to SSH into a container yourself. This includes the username, password, IP address, and port. WARNING: You should only use this command if there is a problem with the tool's ability to SSH into the container. The container_name is the name of the container whose SSH information gets output.

2.2.17 Server Halt

```
$ jab server-halt
```

Listing 2.17: Server Halt command

Gracefully halts the internal server that manages your container. This gracefully stop all containers that are currently running.

2.2.18 Ping

```
$ jab ping
```

Listing 2.18: Ping command

Ensures that the server is running and prints the amount of time it takes to contact the server.

2.2.19 Update

```
$ jab update
```

Listing 2.19: Update command

Updates your installation of Jabberwocky if a new version is detected. This searches for current release in the official GitHub repository. It will then stop the server and install the newest version.

2.2.20 Panic

```
$ jab panic
```

Listing 2.20: Panic command

Stops the internal server ungracefully. WARNING: Only use this if server-halt fails. This could create issues with the installation of the tool and containers.

2.2.21 Version

```
$ jab version
```

Listing 2.21: Version command

Prints the version of the tool that is currently running.

2.3 Example Usages

2.3.1 Installing and Starting a Container

Suppose that you have an archive on your computer installed at ./folder/archive.tar.gz, and you want to install it as a container named the_container. You would use the following commands:

```
$ jab install ./folder/archive.tar.gz the_container
Installing the_container. This may take several minutes... Done!
$ jab start the_container
Starting the_container... Done!
$ jab run the_container echo Hello
Hello
$ jab stop the_container
Done.
```

Listing 2.22: Installing and Starting Container

2.3.2 Moving a File to the Container

We will create a new file at ./my_file.txt. Then, we will move it to the container (into the root directory) and ensure that it has the right content.

```
$ cat > ./my_file.txt
Hello World!
$ jab start the_container
Starting the_container... Done!
$ jab send-file the_container ./my_file.txt
Copying './my_file.txt' -> '~'... Done!
$ jab run the_container cat ~/my_file.txt
Hello World!
$ jab stop the_container
Done.
```

Listing 2.23: Send File

2.3.3 Moving a File From a Container

Suppose that there is a file at ./container_file.txt in the container with the content "Hello World!". We will move it to ./folder/local_file.txt on the local system.

```
$ jab start the_container
Starting the container... Done!
$ jab get-file the_container ./container_file.txt ./folder/
    local_file.txt
Copying './container_file.txt' -> './folder/local_file.txt'... Done
    !
$ cat ./folder/local_file.txt
Hello World!
$ jab stop the_container
Done.
```

Listing 2.24: Get File

Chapter 3

For Instructors

This chapter provides information that instructors will need to know in order to incorporate this tool into their coursework.

In section 3.1, we explain how to use this tool to create a new container for your course. In section 3.2, we explain how to send containers to students for them to use during the course.

3.1 Container Builder

The container build process allows for the creation of containers with customized configurations. Currently, container building is only supported on Linux-based operating systems.

The build process requires the following tools: QEMU, bash, sudo, debootstrap, chroot, libguestfs-tools, awk, and sed.

In order to build containers which emulate a foreign architecture, the host system must be capable of emulating the desired architecture. On most Linux-based systems, this is automatically configured when QEMU is installed via the system's package manager.

3.1.1 build-init

The build process is run within the context of a directory structure which must be initialized with the build-init command as follows:

```
$ jab build-init # Initializes the current directory, or..
$ jab build-init /path/to/desired/directory
```

Listing 3.1: build-init

Once the command has completed, the following structure will be present in the specified directory:

```
[init'd directory]
├── build
│   └── temp
├── packages
├── resources
├── scripts
└── manifest.json
```

- The **build/** directory will contain the results of the build after the `build` command is finished running.
- The **build/temp/** directory contains temporary files used during the build process, and can be safely deleted once the build process has completed via the `build-clean` subcommand.
- Any .deb files found in the **packages/** will be installed into the container during the build process.
- Any files found in the **resources/** directory will be copied into the container's virtual filesystem in the default user's home directory.
- Any files found in the **scripts/** directory will be executed inside the container during the build process. All scripts must be prefixed with a valid interpreter directive (ie, `#!/bin/bash`) or they will fail to execute.
- A default `manifest.json` will be generated.

3.1.2 build

The build command will build the container defined by a provided directory. The directory structure must match the one specified in Section 3.1.1. The build command must be run with root privileges.

```
$ jab build # use the current working directory, or..
$ jab build /path/to/desired/directory
```

Listing 3.2: build

The build process perform the following tasks in order:

- Generate and configure the base system.
- Install any packages specified in the `aptpkgs` field of the `manifest.json`.
- Copy all files in the **resources/** directory into the default user's home directory on the virtual filesystem.
- Install any .deb files in the **packages/** directory.

- Execute any scripts in the **scripts/** directory. Scripts specified in the `scriptorder` field of the `manifest.json` will be executed first, then all unlisted scripts will be executed in an arbitrary order.
- Export the new container to a file.

Once the build command has completed, the following files will be produced in the **build/** directory.

- The container archive file will be stored in `build/jcontainer.tar.gz` or `build/jcontainer.tar` depending on whether or not the `--uncompressed` flag is provided.
- The `build/failed_scripts.txt` file lists all of the scripts in the **scripts/** directory that exited with a non-zero exit code during the build process. If the `build/failed_scripts.txt` file does not exist, that indicates that all scripts exited with an exit code of zero.
- The `build/package_errors.txt` file contains any text output to stderr by `apt` during the part of the build process where the packages stored in **packages/** are installed. Please note that the presence of an error message from apt does not necessarily indicate that a package failed to install. In the event that this occurs, you should check the container produced at the end of the build process manually to verify if this is the case.

3.1.3 build-clean

The `build-clean` command will clean any temporary files generated during the build process.

```
$ jab clean # use the current working directory, or..
$ jab clean /path/to/desired/directory
```

Listing 3.3: build-clean

3.1.4 manifest.json

- **manifest**: The version of the manifest specification. The version this guide describes is 0. It is not recommended that you change this value.
- **arch**: The architecture of the container.
- **memory**: The maximum amount of memory the container can allocate to itself in megabytes.
- **hddmaxsize**: The maximum amount of permanent storage space the container can allocate to itself in gigabytes.
- **hostanme**: The container's hostname.
- **release**: The codename of the Debian version to use as a base.

- **portfwd**: A list of port forwarding rules, which will forward a port on the container's virtual network to one the host system's local network. For example, the rule `[80, 20080]` would forward the container's virtual port 80 to the host's local port 20080.
- **aptpkgs**: Packages to be fetched and installed from the default Debian repositories.
- **scriptorder**: The order to execute the scripts in the **scripts/** directory. Scripts specified in the `scriptorder` field will be executed first, then all unlisted scripts will be executed in an arbitrary order.
- **password**: The password of the container's default user. It is not recommended that you change this value.

3.2 Sending Containers to Students

There are two ways to send containers to your students. The first way is to use repositories, information on which can be found in Section 4.1. The second way is to send archive files to students. More information on archives can be found in Section 2.1. To create an archive from a container, see Section 2.2.

Chapter 4

Other Features

This chapter provides information on some of the other features of our tool that, while are not necessary in order for it to be used, can provide extra functionality for certain situations.

In section 4.1, we describe repositories, which are a way to provide a central server where you can upload and download course containers. In section 4.2, we describe some of the pre-made containers that have already been built for this tool.

4.1 Repositories

A repository represents a central server where archive files are stored. The server allows faculty to easily upload containers using their container manager and students to easily download containers. Thus, faculty can easily dispense containers to students instead of having to send students archive files manually.

4.1.1 Setting Up Repositories

A repository is a Flask server. We only provide the source code for the repositories, so the user will have to host it on a server themselves. Once you have the server setup, there are two commands that you will need to know for setting up your Flask server.

First, you will want to set the path where archives are stored. This location will be where archives can be downloaded from and where archives will be uploaded to. The following command will allow you to set this path (to the value path):

```
1 $ flask set-path [path]
```

Listing 4.1: Setting Path

Second, you will likely want to start the server. The following command will start the Flask server:

```
$ flask --app main run
```

Listing 4.2: Start Server

We will provide one final note about setting up a repository. The source code of the repository contains a file called Auth.py. This file contains a function called auth, which takes two parameters: username and password. When a user attempts to upload to a repository, this function is called with their username and password. If it returns true, the upload occurs. If not, the upload does not occur. By default, this function will always return false (essentially disabling the upload feature). It is the responsibility of the user to create their own auth function. The user is encouraged to integrate this with TRACKS login. We note that the upload feature is not necessary for repositories to work as server owners could manually add the archive files to the specified path.

4.1.2 Uploading to Repositories

There are two ways of adding to a repository. You can either add a file to the archive path of the repository server, as described in the previous section, or you could use the container manager.

Suppose that you have a container installed in your container manage, and you would like to upload an archive of that container to a repository server, then you will want to use the upload command.

```
$ jab upload [container_name] [repo_url]
```

Listing 4.3: Upload command

The container_name will be the name of the container that you wish to upload, and the repo_url will be the link the home page of the repository server. This command will prompt you to input a username and password. You will be authenticated by the server based on this information. If the manager of the repository server did not create a custom Auth.py file, then uploading will always fail to authenticate you.

4.1.3 Downloading from Repositories

Downloading from repositories is slightly more complicated than uploading. Your container manager will store a list of repositories internally, which it will use to find a particular container that you are looking for. With each repository, your container manager will store a list of archive files that are available from that repository. This list is cached, so you will need to occasionally run the update-repo function.

There are three commands that you will need to use when downloading from repositories: add-repo, update-repo, and download.

```
$ jab add-repo [repo_url]
```

Listing 4.4: Add repository command

This command will add a repository server to your internal list of available repository servers. When it does this, it will also query the server and save the list of archive files that the server is currently holding. The repo_url is the address of the repository server that you are adding.

```
$ jab update-repo {repo_url}
```

Listing 4.5: Update repository command

This will query a given repository to determine what archive files are available on the server. It will then update its internal list of those archive files. The repo_url is the address of the repository server that you are updating. If this is not specified, it will update all repositories in your container manager.

```
$ jab download [archive_name] [container_name]
```

Listing 4.6: Download command

Downloads an archive from the list of repositories and installs it to your container manager. It will search through all of the repositories saved to your manager and check their internal list of archives until a match is found. It will prompt you to accept the download. The archive_name is the name of the archive file you wish to download. The container_name is the name you want the new container installed on your system to be called.

4.2 Pre-Made Containers

4.2.1 Compiler Theory

This container was created to be used with the CSE 4251 Compiler Theory course. It contains the ability to compile to, assemble to, load to, and run SPARC binary files. It also contains Java and JavaCC. The containers also have access to the malloc function, for allocating memory from binaries.

This container is actually built with an architecture of x86_64, but it has QEMU running inside (QEMU running in QEMU), which allows SPARC binaries to be run as if they were being run natively. The container also has the special commands sparc-linux-gcc, sparc-linux-as, and sparc-linux-ld for running the SPARC versions of gcc, as, and ld (respectively).

The container is running Java 17.0.4 and JavaCC 5.0.

4.2.2 Programming Language Concepts

The Programming Language Concepts container contains a standard Debian bullseye base along with the following tools:

- **gcc** and **g++** for compiling C and C++ programs.
- **gnatmake** for compiling Ada programs.
- **go** for compiling Go programs.

- **rustc** and **cargo** for compiling Rust programs.
- **gfortran** for compiling Fortran programs.
- **ghc** for compiling Haskell programms.
- **ghci** for an interactive Haskell shell.
- **prolog** for an interactive SWI Prolog shell.

4.2.3 Operating Systems Concepts

The Operating Systems Concepts container contains a standard Debian bullseye base along with a base OS/161 setup and all relevant tools required to build OS/161 located in the `$HOME/os161` directory.