funtoo-metatools

Release 0.1

Oct 28, 2020

Contents:

Funtoo metatools is a framework of powerful technologies created by Daniel Robbins, the creator of Gentoo Linux, which allows you to create and maintain an advanced Gentoo-based Linux distribution.

Funtoo Linux uses `metatools` extensively to perform all its automated functionality, including auto-creation of ebuilds, auto-creation of "kits" and "meta-repo" from upstream repositories or locally-forked packages, CDN mirroring of distfiles (source code artifacts) and advanced analytics. Basically, it does 'all the things' related to creating and maintaining Funtoo Linux.

You can use `metatools` as part of your own distribution infrastructure, or leverage it as you contribute to Funtoo Linux.

See *What is Metatools?* for more details on what is and what it can do for you. Learn more about the pluggable technology upon which `metatools` is based by reading *POP Framework*.

CHAPTER 1

Why is this Free?

This technology, which contains essentially all the technology that comprises Funtoo Linux, is being released under an Open Source license (*Apache Software License, version 2*) along with easy-to-understand documentation, based on the belief that software should be free to use and modify, and easy to understand, use and extend. I want this technology to not just be 'Open Source' but be 'open' in the sense that it can be used, understood and enjoyed by others.

CHAPTER 2

Beyond the License

Open Source is more than a license. Free software is more than just source code. It is a philosophy of developing software that benefits greatly from true collaboration and openness. Qualities such as honesty, integrity, transparency of intent, not pursuing your own ambition in a way that is harmful to others or their endeavors, and a sincere desire to help and give back are the attributes that form the foundation of Open Source and Free Software. These qualities result in people from around the world working together, and ultimately sharing source code.

Said another way – You can use this software, and even abide by its licensing terms, and still potentially undermine the pillars of true collaboration.

I created Gentoo Linux and now Funtoo Linux with the to collaborate with you, and to have fun by working together towards common interests. I ask that you consider your own ethical commitment not just to the license, but to the projects that you choose to use, participate in and support.

I will defend your right to fork, or simply take this code and do your own thing without giving anything in return, because the source code license allows you to do that. But choosing to spurn collaboration with upstream projects can be a frivolous exercise which negatively impacts the collaborative ecosystem that sustains Open Source and free software. We should all be trying our best not just to use free software, nor merely benefit personally from it, but to work together in the spirit of Open Source, as much as we are able.

At its heart, Open Source is more than a license or source code. It is a shared code *of ethics* which governs our behavior and guides our actions towards one another:

> **NOW this is the Law of the Jungle — as old and as true as the sky;**
> **And the Wolf that shall keep it may prosper, but the Wolf that shall break it must die.**
>
> **As the creeper that girdles the tree-trunk the Law runneth forward and back —**
> **For the strength of the Pack is the Wolf, and the strength of the Wolf is the Pack.**
>
> — Rudyard Kipling, The Jungle Book

We should all be seeking to strengthen our "pack". This is the true meaning of collaboration, which is the heart of Open Source and free software.

2.1 What is Metatools?

Metatools is an advanced framework of powerful technologies to allow the auto-creation of Gentoo ebuilds, maintenance of an independent fork of Gentoo or Funtoo, or even building a non-Gentoo distribution. It contains several technology components, including:

- `doit`: A YAML and Jinja-based package pure Python auto-generation engine. See *Auto-generation*.

  This is used for auto-generation of ebuilds. Auto-generation can mean one or more of the following:

  - Querying upstream APIs such as GitHub, GitLab, PyPi and Web sites to find latest versions of packages.
  - Leveraging Jinja to create ebuilds using templates.
  - Downloading and extracting source code and inspecting build scripts to create ebuilds based on their contents. (example: xorg-proto).

- `merge-kits`: A YAML-based mechanism to build and update a "meta-repo" (ports tree) split into logical "kits", assembling these kits from either your own sets of packages or from third-party overlays and repositories. `merge-kits` leverages multi-threading as well as `doit` to perform auto-generation of ebuilds on a distro-wide scale. See *Working with Meta-Repo*.
- `fastpull`: An efficient HTTP spider and CDN infrastructure, used to download and manage source code artifacts, implement source code traceability, and seed a content distribution network for download of source code.
- `deepdive`: Package analytics functionality. This allows the contents of Funtoo or your distribution to be analyzed via MongoDB queries to understading relationships between packages.

2.2 POP Framework

Funtoo-metatools uses Thomas Hatch's POP (Plugin-Oriented Programming) framework as its foundational paradigm for code organization. POP (think 'OOP' but with a 'P') is a next-generation framework that encourages code maintainability and extensibility, and successfully solves code bloat problems inherent in application of OOP paradigms at scale. Here are some resources related to pop:

- Introduction to Plugin Oriented Programming
- POP (PyPi page)

2.3 POP and Portage

I am really excited about POP because it helps to solve quite a few problems that the current Portage (Gentoo package manager) code base suffers from.

Portage is not unique in this regard – it's been around for a while, and has had a ton of functionality bolted on which has made it hard to improve and adapt.

In fact, many people who have tried to hook into Portage APIs get frustrated and create their own code to try to do what they want – because Portage's code is set up almost exclusively for the purpose of implementing the functionality of the `ebuild` and `emerge` commands – and not really to be leveraged by others.

This has been a source of over a decade of frustration for me. After all, I can remember when `ebuild` was simply a 150-line bash script that I wrote. And surprisingly, it implemented all the necessary functions to build packages. It was very minimalistic. Now, portage consists of over 1000 *source code files* and 135,000 lines of Python code. That's just really big.

This isn't really the "fault" of Portage as much as it is the result of being a project that has been around for a while, and it has grown organically as new features and capabilities have been added.

You would think that all this new code has resulted in a powerful API that other people can use to do amazing things. But one of the failings of OOP (Object Oriented Programming) at scale is that it creates complex heirarchies of inter-dependent classes that don't really function in a stand-alone fashion. So while the Portage code base enables `emerge` and `ebuild` to function, it is not being leveraged by other tools. It was not really designed to do this.

Plugin-oriented programming helps to fix this. It turns the often insular OOP paradigm upside-down and provides the technology to not only extend funtoo-metatools easily, but also allow *your* tools and utilities to leverage funtoo-metatools' internal code easily. So we're not just building a tool – we're building a modern community framework that you can both contribute to and leverage.

Also, please note – my intent in mentioning Portage is not to pick on it, or those who have maintained it over the years whose efforts I appreciate, but rather to explain why I am so excited about building metatools and creating a framework that can be more successfully leveraged by the Open Source community. My long-held desire to continue to improve Portage has been restrained by the very structure of the source code that has evolved within it. Originally, Portage was a tool that solved problems. It created new paradigms. It has evolved into something that while still cool, also enforces a paradigm that is hard to change and adapt to new problems.

Due to our use of POP, much of metatools functionality is extensible via plugins. Plugins can be used to enhance the core functionality of the tool in a modular 'plug-n-play' way, reducing code bloat. POP also encourages a simple, microservices-style archictecture within the code itself. All this is very good for a complex problem like the automation of updates of packages for the world's source code.

So, remember – plugin-oriented programming allows you to do two things. First, it allows you to easily *extend* funtoo-metatools. Second, through the magic of dynamic plugin registration, it allows you to easily *leverage* the power of funtoo-metatools within your own applications. It also provides a really clean paradigm for adding functionality to funtoo-metatools over time, avoiding complex internal interdependencies that make code harder to maintain and adapt.

2.4 Installation

These instructions asssume you are using Funtoo Linux but should be easy to adapt to other distributions.

2.4.1 Installing Latest Official Release

Funtoo-metatools is easy to install. On Funtoo, it can simply be emerged:

```
# emerge metatools
```

Recent version of metatools also require MongoDB to be installed and running locally. MongoDB is used by the `doit` command to store distfile integrity hashes, as well as for maintaining a cache of HTTP API requests for more resilience related to network interruption. It is also used by `deepdive` to provide package analytics functionality.

MongoDB can be installed and started on Funtoo Linux as follows:

```
# emerge mongodb
# rc-update add mongodb default && rc
```

Alternatively you can use `pip3` to pull it from PyPi:

```
$ pip3 install --user funtoo-metatools
```

If you would like to create an isolated virtual environment for funtoo-metatools, you can use virtualenv as follows:

```
$ emerge virtualenv
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv) $ pip install funtoo-metatools
```

From this point forward, whenever you want to use the virtualenv, simply source the activate script to enter your isolated python virtual environment.

2.4.2 Installing Latest Development Sources

If you need the absolute latest features, you may want or need to install `metatools` from git master. To do this, you will want to clone the repository locally:

```
# cd ~/development
# git clone ssh://git@code.funtoo.org:7999/~drobbins/funtoo-metatools.git
```

Next, you will want to add something similar to the following to you shell init script, such as your `./.bashrc` file:

```
export PATH=$HOME/development/funtoo-metatools/bin:$PATH
export PYTHONPATH=$HOME/development/funtoo-metatools
```

When these settings are active, your current shell will be able to find the binaries such as `doit` as well as the python modules in the live git repository.

Finally, you will want to ensure that all dependent modules are installed. On Funtoo, this can be accomplished via `emerge --onlydeps metatools`.

2.5 Auto-generation

`metatools` includes the `doit` command, which implements "auto-generation" of ebuilds. But what exactly is "auto-generation"?

In its broadest sense, "auto-generation" is the high-level creation of ebuilds. This can involve any number of advanced capabilities, such as querying GitHub or GitLab to find the latest version of a package, actually fetching source code and looking inside it, or using Jinja to generate ebuilds using templates. Typically, multiple approaches are used together.

We use these capabilities to *reduce the manual labor required to maintain packages*. These capabilities exist to give us *leverage* over the complex world of software so that we can automate as much as possible, so we can do more with less.

2.5.1 Running Auto-Generation

To actually use these tools to auto-generate ebuilds, it is recommended that you check out the kit-fixups repository, which is the master repository of Funtoo Linux. This repository is organized into directories for each kit, and sub-directories for each kit version, with `current` often used as a sub-directory name for kit version. So, for example, if we wanted to auto-generate all ebuilds in `core-kit/current`, we would do this:

```
$ cd development
$ git clone https://code.funtoo.org/bitbucket/scm/core/kit-fixups.git
$ cd kit-fixups/core-kit/curated
$ doit
```

In the above example, it will see that the directory `kit-fixups/core-kit/curated` is an overlay that contains categories, and it will look inside it for all "autogens" and execute them.

When `doit` runs, it will determine its context by looking at the current working directory, similar to how the `git` command will find what git repository it is in by looking backwards from the current working directory. It will then fire off auto-generation in the current directory, looking in the current directory *and any sub-directories* for all autogens, and will execute them. You will see a lot of `....` being printed on the screen, which means that files are being downloaded. What is actually happening is that `doit` is querying Web APIs like GitHub and GitLab to find the latest versions of packages, and then downloading the source code tarballs (in `metatools` vernacular: "artifacts") for these packages, and a period is printed for each block of data received to show progress. Often times, multiple artifacts are being downloaded at the same time. Then, as the artifacts are received, `doit` creates ebuilds for these packages, and also creates `Manifest` files referencing the SHA512 and other digests of the downloaded Artifacts. You end up with ebuilds that you can test out by running `ebuild foo-1.0.ebuild clean merge`.

Where are these "autogens"? They are `autogen.py` files that exist in the repository. Think of our autogens as plug-ins, written in Python and leveraging the *POP Framework*, that contain a `generate()` function which can generate one or more ebuilds using the `metatools` API. The `metatools` API, which we'll look at in a bit, is an extensible API that lets us query Web APIs, use Jinja and perform other neat tricks to generate ebuilds.

In addition to raw autogens, there are also `autogen.yaml` files which allow for creation of ebuilds *en masse*. In the YAML, you specify an autogen (also called a metatools "generator") plus packages and package-specific metadata to feed to that generator. When you feed package data to a generator, it spits out ebuilds. This is both highly efficient (it's fast) and also a nice way to generate ebuilds with little or no redundant code. `metatools` contains a number of built-in generators that can be used with the YAML system, such as generators that build ebuilds for Python packages on PyPi.

Go ahead and poke around inside kit-fixups and look at the `autogen.py` and `autogen.yaml` files. You'll begin to get a sense for what they look like and and inkling of how everything works.

Also type `git status`. You should see that a bunch of ebuilds (along with `Manifest` files) were created. These files are *not* added to git. They simply sit in your local repo, and you can blow them away by running:

```
$ git clean -fd
```

When doing developent, we actually *do not* want to commit the auto-generated ebuilds themselves to kit-fixups – we just want to commit the autogens (`autogen.py` and `autogen.yaml`.) There is a separate step, peformed by the `merge-kits` command, which updates the meta-repo and will commit the generated ebuilds to kits which are then pushed out to users. But for `kit-fixups`, we're doing development, not updating the tree, so we just want to commit the autogens.

2.5.2 Developing Auto-Generation Scripts

Now that we've covered how to execute auto-generation scripts, let's take a look at creating them.

Basic Stand-Alone Layout

The simplest form of auto-generation is called *stand-alone* auto-generation. Stand-alone auto-generation scripts have the name `autogen.py` and can be located inside a catpkg directory – at the same level that you would place ebuilds. Typically, you would also create a `templates/` directory next to `autogen.py`, containing template files that you use to create your final ebuilds. For example, if we were doing an autogen for a package called `sys-apps/foobar`, which is a "core" system package, we would:

1. Create an `autogen.py` file at `kit-fixups/curated/sys-apps/foobar/autogen.py`
2. Create a `kit-fixups/curated/sys-apps/foobar/templates/foobar.tmpl` file (a template for the ebuild.)

The Generator

The `autogen.py` script is, as you might guess, a python file. And it is actually treated as a *plugin* (see *POP Framework*) which gives it a special structure. The auto-generation function that gets called to do all the things is called `generate()` and should be defined as:

```
async def generate(hub, **pkginfo):
```

Here is a full example of an `autogen.py` that implements auto-generation of the `sys-apps/hwids` package:

```
#!/usr/bin/env python3

async def generate(hub, **pkginfo):
  github_user = "gentoo"
  github_repo = "hwids"
  json_list = await hub.pkgtools.fetch.get_page(
      f"https://api.github.com/repos/{github_user}/{github_repo}/tags", is_json=True
  )
  latest = json_list[0]
  version = latest["name"].split("-")[1]
  url = latest["tarball_url"]
  final_name = f'{pkginfo["name"]}-{version}.tar.gz'
  ebuild = hub.pkgtools.ebuild.BreezyBuild(
      **pkginfo,
      github_user=github_user,
      github_repo=github_repo,
      version=version,
      artifacts=[hub.pkgtools.ebuild.Artifact(url=url, final_name=final_name)],
  )
  ebuild.push()
```

The `doit` command, when run in the same directory in the `autogen.py` or in a parent directory that is still in the repo, will find this `autogen.py` file, map it as a plugin, and execute its `generate()` method. This particular auto-generation plugin will perform the following actions:

1. Query GitHub's API to determine the latest tag in the `gentoo/hwids` repository.
2. Download an archive (called an *Artifact*) of this source code if it has not been already downloaded.
3. Use `templates/hwids.tmpl` to generate a final ebuild with the correct version.
4. Generate a `Manifest` referencing the downloaded archive.

After `autogen.py` executes, you will have a new `Manifest` file, as well as a `hwids-x.y.ebuild` file in the places you would expect them. These files are not added to the git repository – and typically, when you are doing local development and testing, you don't want to commit these files. But you can use them to verify that the autogen ran successfully.

The Base Objects

Above, you'll notice the use of several objects. Let's look at what they do:

`hub.pkgtools.ebuild.Artifact` This object is used to represent source code archives, also called "artifacts". Its constructor accepts two keyword arguments. The first is `url`, which should be the URL that can be used to download the artifact. The second is `final_name`, which is used to specify an on-disk name if the `url` does not contain this information. If `final_name` is omitted, the last part of `url` will be used as the on-disk name for the artifact.

hub.pkgtools.ebuild.BreezyBuild This object is used to represent an ebuild that should be auto-generated. When you create it, you should pass a list of artifacts in the `artifacts` keyword argument for any source code that it needs to download and use.

These objects are used to create a declarative model of ebuilds and their artifacts, but simply creating these objects doesn't actually result in any action. You will notice that the source code above, there is a call to `ebuild.push()` – this is the command that adds our `BreezyBuild` (as well as the artifact we passed to it) to the auto-generation queue. `doit` will "instantiate" all objects on its auto-generation queue, which will actually result in action.

What will end up happening is that the `BreezyBuild` will ensure that all of its source code artifacts have been downloaded ("fetched") and then it will use this to create a `Manifest` as well as the ebuild itself.

pkginfo Basics

You will notice that our main `generate` function contains an argument called `**pkginfo`. You will also notice that we pass `**pkginfo` to our `BreezyBuild`, as well as other additional information. What is this "pkginfo"? It is a python dictionary containing information about the catpkg we are generating. We take great advantage of "pkginfo" when we use advanced YAML-based ebuild auto-generation, but it is still something useful when doing stand-alone auto-generation. The `doit` command will auto-populate `pkginfo` with the following key/value pairs:

name The package name, i.e. `hwids`.

cat The package category, i.e. `sys-apps`.

template_path The path to where the templates are located for this autogen, i.e. the `templates` directory next to the `autogen.py`

While this "pkginfo" construct doesn't seem to be the most useful thing right now, it will soon once you start to take advantage of advanced autogen features. For now, it at least helps us to avoid having to explicitly passing `name`, `cat` and `template_path` to our `BreezyBuild` – these are arguments that our `BreezyBuild` expects and we can simply "pass along" what was auto-detected for us rather than specifying them manually.

Querying APIs

It is not required that you query APIs to determine the latest version of a package to build, but this is often what is done in an `autogen.py` file. To this end, the official method to grab data from a remote API is `hub.pkgtools.fetch.get_page()`. Since this is an `async` function, it must be `await``ed. If what you are retrieving is JSON, then you should pass ``is_json=True` as a keyword argument, and you will get decoded JSON as a return value. Otherwise, you will get a string and will be able to perform additional processing. For HTML data, typically people will use the `re` (regular expression) module to extract data, and `lxml` or `xmltodict` can be used for parsing XML data.

There is also a `refresh_interval` keyword argument which can be used to limit updates to the remote resource to a certain time interval. For example, this is used with the `brave-bin` autogen to ensure that we only get updates every 5 days (they update the Brave browser daily and this update interval is a bit too much for us):

```
json_dict = await hub.pkgtools.fetch.get_page(
  "https://api.github.com/repos/brave/brave-browser/releases", is_json=True, refresh_
↪interval=timedelta(days=5)
)
```

HTTP Tricks

Sometimes, it is necessary to grab the destination of a HTTP redirect, because the version of an artifact will be in the redirected-to URL itself. For example, let's assume that when you go to `https://foo.bar.com/latest.tar.`

gz, you are instantly redirected to https://foo.bar.com/myfile-3002.tar.gz. To grab the redirected-to URL, you can use the following method:

```
next_url = await hub.pkgtools.fetch.get_url_from_redirect("https://foo.bar.com/latest.
↪tar.gz")
```

next_url will now contain the string https://foo.bar.com/myfile-3002.tar.gz, and you can pull it apart using standard Python string operators and methods to get the version from it.

Note that both the Zoom-bin autogen and Discord-bin autogen use this technique.

Using Jinja in Templates

Up until now, we have not really talked about Templates. Templates contain the actual literal content of your ebuild, but can include Jinja processing statements such as variables and even conditionals and loops. *Everything passed to your ''BreezyBuild'' can be expanded as a Jinja variable.* For example, you can use the following variables inside your template:

{{cat}} Will expand to package category.

{{name}} Will expand to package name (without version).

{{version}} Will expand to package version.

{{artifacts[0].src_uri}} Will expand to the string to be included in the SRC_URI for the first (and possibly only) Artifact.

SRC_URI="{{artifacts|map(attribute='src_uri')|join(' ')}}" Will expand to be your full SRC_URI definition assuming you don't have any conditional ones based on USE variables.

It's important to note that in some cases, you will not even need to use a single Jinja-ism in your template, and can simply have the entire literal ebuild as the contents of your template. The Discord-bin autogen template is like this and simply contains the contents of the ebuild, because the only thing that changes between new Jinja versions is the filename of the ebuild, but not anything in the ebuild itself. So we don't need to expand any variables.

But when we get into more advanced examples, particularly YAML-based auto-generation, Jinja tends to be used more heavily.

Here are some other Jinja constructs you may find useful:

```
{%- if myvar is defined %}
myvar is defined.
{%- else %}
myvar is not defined.
{%- endif %}

{%- if foo == "bar" %}
This text will only be included if the variable "foo" equals the string "bar".
{%- elif foo == "oni" %}
Hmmm... foo is oni?
{%- endif %}

{%- for file in mylist %}
{{file}}
{%- endfor %}
```

You can see that Jinja gives you a lot of power to generate the final representation of the ebuild that you want. Remember that you can always pass new keyword arguments to the constructor for BreezyBuild and then access them in your templates. For more information on what Jinja can do, browse the official Jinja Documentation or look in the kit-fixups repo for interesting examples.

Using Multiple Templates or BreezyBuilds

As mentioned earlier, you can place templates in the *templates/* directory next to your autogen, and by default, the `BreezyBuild` will use the template with the same name as your package. To change this, you can pass the `template="anothertemplate.tmpl"` keyword argument to your `BreezyBuild` or pass a different `name` to your `BreezyBuild` (`name` is normally part of the `**pkginfo` dict.) You might want to do this if you are using your `autogen.py` to generate *more than one* ebuild – which is perfectly legal and supported. In this case, you will want to vary the `name` and/or `cat` arguments that get passed to `BreezyBuild` (these typically come via `**pkginfo`) to specify a new package name and/or category. Remember to call `.push()` for every ebuild you want to generate. See the Virtualbox-bin Autogen for an example.

Introspecting Inside Artifacts

You may be wondering if it is possible to grab a source tarball, look inside it, and parse things like `Makefile` or `meson.build` files to base your build steps on stuff *inside* the Artifact. Yes, this is definitely possible. To do it, you will first want to define an `Artifact` all by itself, and then call its `ensure_fetched()` or `fetch()` async method. You can then unpack it and inspect its contents:

```
import os
import glob

async def generate(hub, **pkginfo):
  my_artifact = Artifact(url="https://foo.bar.com/myfile-1.0.tar.gz")
  await my_artifact.ensure_fetched()
  my_artifact.extract()
  for meson_file in glob.iglob(os.path.join(my_artifact.extract_path, "*/meson.build
↪"):
    ...
  my_artifact.cleanup()
```

See our xorg-proto Autogen for an example of this. It downloads `xorg-proto` and introspects inside it to generate a bunch of stub ebuilds for each protocol supported by `xorg-proto`.

2.6 Working with Meta-Repo

`metatools` contains the `merge-kits` command which is a distribution maintainer tool for creating and updating `meta-repo`. It can also be used for developers for generating a local `meta-repo` based on the contents of `kit-fixups` for local testing.

This script works by sourcing ebuilds from various overlays, and combining them using special algorithms to yield the kits you use. A meta-repo is also generated, which points to the specific kits generated that are designed to work together.

Funtoo Linux uses `merge-kits` in "production mode" (with the `--prod` option) to update Funtoo Linux – both `meta-repo` and the kits that end up on your system.

But it's also possible for developers to use `merge-kits` locally, to test local changes by generating a local `meta-repo`.

2.6.1 Definitions

Before diving in, it would be good to define some terms for people new to Funtoo Linux. This is worth reading even if you know these terms as I'm going to set some context for the rest of this documentation here:

ebuild An ebuild is an individual build script for Gentoo or Funtoo Linux, and ends with an `.ebuild` extension, and has a package name and version in its filename. For example: `bash-5.0.ebuild`.

repo or overlay Ebuilds are generally always organized into a "repo" (repository) or "overlay", which are essentially the same thing. This is a special directory tree that organizes ebuilds in the following structure: `my-repo/category/package/package-version.ebuild`. Gentoo Linux has a monolithic "Portage tree" which is organized in this way and contains around 30,000 ebuilds. In Funtoo, we try to organize this tree into logical "kits" (we'll get to that definition soon.) There are also very many third-party overlays that exist in the Gentoo ecosystem, which can be added to your system to add new packages.

catpkg Short-hand for "category-package", this is a specifier to uniquely identify a package by category. Example: `app-shells/bash`. It doesn't contain a version.

kit Used by Funtoo Linux, kits are official overlays of the Funtoo Linux project, with ebuilds organized by theme. For example, Funtoo Linux has a `gnome-kit` which contains all the ebuilds related to GNOME. Kits have a few minor differences from standard Gentoo overlays. First, we make some effort to ensure that a `catpkg` only exists in a single kit. Secondly, kits do have separate named git branches, such as `4.20-release`.

meta-repo Typically living at `/var/git/meta-repo`, "meta-repo" is Funtoo's equivalent of Gentoo's "Portage Tree" or package database. A user of Funtoo Linux will grab the latest packages by running `ego sync`, which will update the contents of `meta-repo` using `git`. `meta-repo` is actually a very small git repository that only contains metadata that references the kits that make up a `release` of Funtoo Linux, as well as what the official branch is for that release, and what commit is the most recent.

kit-fixups Kit-fixups is the 'master' repository for Funtoo Linux which contains YAML definitions of the Funtoo Linux release, kits, as well as our locally-maintained forked `catpkgs` for all our kits.

2.6.2 Developer First Run

A developer might use the `merge-kits` command to generate a local version of `meta-repo` for testing. When used this way, you will have a private copy of `meta-repo` and kits. To use it in this way, use the following command:

```
$ merge-kits 1.4-release
```

This command, when run, will do a lot of things:

1. It will clone the official `kit-fixups` repository from `code.funtoo.org` and put it in `~/repo_tmp/source-trees/`.
2. It will look at the YAML in `kit-fixups/foundations.yaml` to determine how to build the `1.4-release` version of Funtoo Linux.
3. It will then clone (or update) any source git repositories specified in `foundations.yaml` and put them in `~/repo_tmp/source-trees/`.
4. A new meta-repo git repository will be created in `~/repo_tmp/dest-trees/meta-repo`, if one doesn't exist.
5. Based on definitions in `foundations.yaml`, it will generate all kits, which will end up in `~/repo_tmp/dest-trees/meta-repo/kits`. This will involve spawning the `doit` command as needed to perform any auto-generation of ebuilds. When autogen is used, the results of autogen (the actual auto-generated ebuilds) *will be committed* to the kit.

Once `merge-kits` completes successfully, you should be able to use the meta-repo you generated by performing the following steps:

```
$ su
# cd /var/git
```

(continues on next page)

(continued from previous page)

```
# mv meta-repo meta-repo.bak
# ln -s /home/user/repo_tmp/dest-trees/meta-repo meta-repo
# ego sync --in-place
```

You can now run `emerge` commands, etc. and your locally-generated meta-repo will be used rather than the official Funtoo one.

2.6.3 Developer Day-to-Day Use

The steps above are a good "first step" for experiencing the wonder of `merge-kits`, but still isn't very useful, because while you generated your own meta-repo, its contents are going to be nearly identical to what Funtoo Linux already offers. The only difference might be that some auto-generated or recently-updated packages might be newer in your version if Funtoo Linux's meta-repo hasn't been updated as recently.

What is usually much more useful is to point `merge-kits` to your own forked `kit-fixups` repository, which then will generate a meta-repo with your own personal changes.

To do this, create a `~/.merge` file with the following contents:

```
[sources]

kit-fixups = ssh://git@code.funtoo.org:7999/~drobbins/kit-fixups.git

[branches]

kit-fixups = my-tweaks
```

You will need to remove any existing `kit-fixups` repo from `~/repo_tmp/source-trees` if it was cloned from another location, but once done, and `merge-kits 1.4-release` is run again, this time `meta-repo` will contain *your* personal changes. This is a great way to test more complex changes that have the potential of blowing things up on a grand scale.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search