title:	Using Arch From an Informal Perspective
license:	General Public License, V2
copyright:	2004, 2005 Canonical Ltd. ../../bazaar-docs/src/FollowingTux.rst:3: (WARNING/2) Cannot extract compound bibliographic field “copyright”.
Author:	James Blackwell <jblack@gnuarch.org>
contributors:	Bob Finney
original-location:	jblack@gnuarch.org--2004/bazaar-docs--initial--1.1

If you are interested in a project that is already using arch, then consider yourself lucky! You are about to embark on some of the easiest revision control you have ever seen in your life.

Before we get started, I am going to define some roles and definitions. We are going to call you, the reader, “jdoe”, while we are going to call the developer of the project that you are interested in “Tux”. Tux is running a free software project called “fishfarm”, which you have recently become interested in. On Tux’s website, he has told you the following:

- He keeps the fishfarm sources in an arch archive
- The archive’s location is at: http://tux.nd/tux@hackingpenguin.nd--2004
- He may or may not have told you his archive’s name is tux@hackingpenguin.nd--2004
- That he keeps a configuration file called “fishfarm” in his dists--tux package.

Tux has recently changed his development model. Once upon a time, he released early and frequently. These days, he’s only releasing two or three times a year. He’s told his users that if they’re interested in following development, then they should use arch as well.

Ok then. So lets use arch. Before we can use arch, we have to tell arch who we are. Don’t worry about getting spammed. This only gets saved locally.

```
$ baz my-id "Joe Blow <jblow@isp.com>"
```

I’m sure that arch is glad to meet you. :) Like many other revision control systems, because you can use an archive, you have to tell arch where to find the archive. We perform this with the register-archive command:

```
$ baz register-archive http://tux.nd/tux@hackingpenguin.nd--2004
Registered archive tux@hackingpenguin.nd--2004
```

Good job. Now, we have a choice. Now, we could either go ahead and get fishfarm, or we can be nosy and poke around in Tux’s archive and see what else he has there. I’ll assume you’re as nosy as me. :

```
$ baz rbrowse tux@hackingpenguin.nd--2004/
tux@hackingpenguin.nd--2004
  dists
    dists--tux
      dists--tux--1
        base-0 .. patch-12

fishfarm
  fishfarm--dev
    fishfarm--dev--1.0
      base-0 .. patch-221
    fishfarm--dev--1.2
      base-0 .. patch-34
```

Don't be scared of it. What you're seeing here is all the various versions of packages that Tux has. We can see that he has a "dists--tux", and a "fishfarm--dev". In fact, he has two fishfarms; the 1.0 series of fishfarm, and the 1.2 series of fishfarm.

That's interesting and all, but how do we see what sort of work he's actually done? That's easy as well. Lets pick a package and a version (what's called a "qualified version"), tack it on to the end of the archive (called the "fully qualified version" and tell rbrowse to show it to us (the -s says to show us the descrption of the patches):

```
$ baz rbrowse -s tux@hackingpenguin.nd--2004/fishfarm--dev--1.2 | head -10
tux@hackingpenguin.nd--2004
  fishfarm
    fishfarm--dev
      fishfarm--dev--1.2
        base-0
          Initial import
        patch-1
          Taught fishfarm to breed even more herring!
        patch-2
          Reduced the chance for a polar bear attack
```

Thats kind of neat, but not really anything special. There are, however, neater things we can do with rbrowse. For example, we could have added --patch-regex "snow", and we would have seen all the patches in which Tux talks about snow.

I digress. Rather than showing you how to browse archives, I should be showing you how to actually get fishfarm. Well, in this case, its pretty easy, because Tux has set up a "config" branch, which we can use to build a system made out of a bunch of packages (though in this case, he has just one package). First, lets get his package:

```
$ baz get tux@hackingpenguin.nd--2004/dists--tux fishfarm
[lots of baz output]
$ cd fishfarm
$ ls
configs/ {arch}
$ ls configs
fishfarm-dev
$ baz buildcfg configs/fishfarm-dev
[lots of baz output]
$ ls
configs fishfarm fishfarm-dev
$ cd fishfarm
```

And there you are! You now have a copy of fishfarm for your use and abuse. You can do all the things you're normally accustomed to in a source tree.

In the next chapter, we'll cover how to maintain local changes. While you grab some coffee, I'm going to get a smoke. See you in a few.

2. Maintaining Local changes

Welcome back. When we last met, we covered how to get a working copy of Tux's work. We covered setting up your id, registering Tux's archive, and using buildcfg to grab his code. Today, we'll cover how to track Tux's project with local changes.

The first thing that you'll have to do is setup an archive. Setting up an archive requires knowing two things: What the name of the archive is, and where you're going to keep it. An archive name has two parts, the "email component" and the "naming component". In your case, your email address is jblow@isp.com, so that will be the email component. For the naming component, we're going to use "fishfarm-2004". Now, we add them together for the whole archive name: jblow@isp.com--fishfarm-2004.

But where ever shall we save the archives? That depends upon your preference. Traditionally, people will make a ~/{archives} dir, but if you really hate using braces, then ~/archives will do fine.

```
$ baz mkdir ~/.archives
$ baz make-archive jblow@isp.com--farm-2004 \
   /home/jblow/.archives/jblow@isp.com--farm-2004
$ baz my-default-archive jblow@isp.com--farm-2004
```

The more observant among you probably noticed that I added an extra command, my-default-archive. This command does exactly what you would think; if we tell arch to get something like "fishfarm--dev", then by default it will go to jblow@isp.com--farm-2004 to get it first.

Congratulations. You now have your own, albeit empty, archive. Don't worry about it being empty; in the very near future we're going to add some new packages, some versions for those packages, and some revisions for that version.

In fact, lets create a package now. As I mentioned before, we'd like to localize Tux's packages so that we can hack on them locally. Lets go ahead and do so now:

```
$ baz branch tux@hackingpenguin.nd--2004/fishfarm--dev--1.2 \
             jblow@isp.com--farm-2004/fishfarm--jblow--1.2
$ echo If you see this then you are paying attention
$ baz branch tux@hackingpenguin.nd--2004/dists--tux--1 \
             jblow@isp.com--farm-2004/dists--jblow--1
```

So, what is a branch? Well, you can almost (but not quite) think of a tag as a sym-link from the last revision of his version (at the time that we do the tag) to the start of our version. From that point on, we can build on that. For example, lets say that tux has 3 revisions in dists--tux--1

```
$ baz rbrowse tux@hackingpenguin.nd--2004/dists--tux--1
tux@hackingpenguin.nd--2004
  dists
    dists--tux
      dists--tux--1
        base-0 .. patch-3
```

And if you look at your archive, you would see something that looked like this:

```
$ baz rbrowse -s
jblow@isp.com--farm-2004
  dists
    dists--jblow
      dists--jblow--1
        base-0
          tag of tux@hackingpenguin.nd--2004/dists--tux--1--patch-3
  fishfarm
    fishfarm--jblow
      fishfarm--jblow--1.2
        base-0
          tag of tux@hackingpenguin.nd--2004/fishfarm--jblow--1.2--patch-34
```

So, now that we've tagged off of Tux, how do we hack on it? For example, we'll have to hack the config file to point at our versions rather than his versions:

```
$ baz get dists--jblow--1 mydists
[lots of baz output]
$ cd mydists
$ vi configs/fishfarm-dev
```

```
[change the config to change the config to point to yours instead
from:  tux@hackingpenguin.nd--2004/fishfarm--dev--1.2
  to:  jblow@isp.com--farm-2004/fishfarm--jblow--1.2
]
```

Now that you've made this change, you still have to commit it to the archive so that it will be there the next time you get. You perform commits with the "baz commit" command. You can get tricky with commit messages by typing something like vi *baz make-log*, but the easiest way to store you changes is with the following:

```
$ baz commit -s"Changing the fishfarm config to point to mine instead"
[bunches of baz output]
```

Sure enough, baz figures out the changes since the last time you committed, and stores them in the archive. You can verify it by either getting another copy of the work tree in a different directory, or by running the "baz logs" command:

```
$ baz logs -s
base-0
  tag of tag of tux@hackingpenguin.nd--2004/dists--tux--1--patch-3
patch-1
  Changing the fishfarm config to point to mine instead
```

By now you're probably starting to think "Hey! I could probably have a whole bunch of working trees, but what happens when I commit to one? Won't the rest of them go out of date?" That is absolutely correct. Whenever a working tree gets out of date, you can catch it up by simply typing "baz update". This will cause baz to undo all of the current changes in the working tree, apply all of the missing changesets one by one, and them reapply the missing changesets.

This is a pretty powerful thing we've done. Even though Tux doesn't know that we exist, we were able to make a localized fork of his code without giving up on having a revision control system at _our_ beck and call. In the next chapter, things start to get more interesting.

3. Merging Tux's work back into you

Of course, while you're doing this work, Tux has been busy as well. He's continued to work on his project and add new patches to his archive by committing them. However, your archive doesn't know about them yet, because you haven't merged his changes.

There are two main ways that you can catch yourself up to Tux. You can either "baz replay" or "baz merge" with his code. There is a simple rule of thumb that you can use when you're trying to decide whether or not to use replay or merge: If you and Tux have both been working in different areas of the code and if Tux hasn't merged from you, then you can use replay. Otherwise, you'll need to use merge. The differences between replay and merge are too deep and significant to cover here; suffice it to say that while "merge" is a much smarter command, "replay" is capable of providing one with a finer granularity of control.

Since Tux has no idea that we even exist at this point, we can safely use replay. The first thing that we have to do is get a clean working copy of our version. As always, we'll perform this step with the "baz get" command. Then, we'll tell baz to try and merge in Tux's latest changes.

```
$ baz get dists--jblow--1 mydists
$ cd mydists
$ baz replay tux@hackingpenguin.nd--2004/dists--tux--1
[ lots of tla output as tla applies Tux's changesets to our
  code ]
```

I should mention an important point again: Make sure that when you are merging somebody else's code, that you do so into a *clean* working copy. Merging local changes and remote changes is a bad

idea. Trust me. What will happen is that down the road, the guy that you merged from is going to try to merge from you, and he's going to get a lot of conflicts. The reason for that is that when you sneak your changes into his changeset, when he merges you, he's not going to have any of that local code you snuck in. So just say "No".

So where were we? Oh yeah. You had just updated your tree to have Tux's new changes. If you're as flighty as I am, you've probably forgotten how the tree has changed. You can always check the status of how a tree has changed by running the following:

```
$ baz status
[baz lists what files were added/deleted/modified, etc]
```

If you want to see how the code actually changed, then a even more useful command is:

$ baz diff | less [less pops up, and you're shown a diff between the last commit this working tree knows about and how the working tree actually looks]

Once you're happy that the replay is really doing what its supposed to do, including resolving any conflicts that happened during merge, then you can go ahead and commit with something like:

```
$ baz commit -L"Merged changes from Tux"
```

You can do this however often you like. You can not merge him for weeks at a time, or you can do it on a daily basis. Its really up to you.

If you're used to the second generation revision control systems, by this point you're probably saying "WOW! I can do that with arch??" Yes, you can, even on a daily basis. But in the next chapter, we're going to hit on some stuff that should knock your socks off... well, maybe not your socks, but at least your shoes.

4. Sharing your work with others

Ok. Now I'm going to set the way-back machine way-forward.... say six months or so, which works out to about 20 years in Internet time. During this time, You've been working on fishfarm. Tux has been working on fishfarm, and you've been merging him. But something inside of you snaps. You're tired of maintaining your local changes, and you decide its time to sluff off your maintenance work on Tux. I'm sorry. Not sluff... "donate". Yeah. You decide to DONATE your code to Tux. :)

The first thing that you have to do in order to donate your work to Tux is to make your archive publicly available. If you're like me, you probably do your work on a laptop. Or perhaps your main work machine is sitting behind NAT of some sort. Maybe you've got some cruddy ISP that only gives you dynamic addresses. If you're particularly unlucky, perhaps all three of them are true. That's fine, because arch can perform a little bit of magic we call mirroring.

The idea behind mirroring is that you can take an archive in one place, and make a copy of it elsewhere. You can also update the mirror whenever it suits you. In order to mirror an archive you own remotely, you need to know two pieces of information: 1. The archive name you want to mirror and 2. Where you want to mirror it. You have a few different ways you can mirror and archive: the local filesystem (no good here, because we've already established that you're machine isn't available to the public), ftp (not such a hot idea, because that means your password is going over the 'net in plaintext), WebDAV (provided you A) have a version of arch that can handle SSL and B) can find a place that allows you to upload via WebDAV), and the old favorite standby, sftp (which is a fancy of way of saying "ssh"). In case you can't tell, I heartily recommend ssh.

For the sake of describing this easily, I'm going to assume that you have setup password-less ssh on some machine out on the 'net. Who knows; you may have setup an account at sourcecontrol.net so that you could push your archive!

The first thing you need to do is to make sure that you can ssh to the machine without having to enter a password. Ok. That's not _really_ required, but if you don't, I don't make any guarantees as to your long term sanity. Usually, the way that you setup ssh for password-less logins goes along the following:

1. $ ssh-keygen -t rsa
2. [follow the prompts, hit enter when it asks for the new password]
3. copy ~/.ssh/id_rsa.pub into .ssh/authorized_keys on the server
   you're trying to ssh to.
4. Make sure that the permissions on .ssh and authorized_keys are
   very limited. If memory serves, you have to make sure that you
   have the equivalent of chmod 600 .ssh .ssh/authorized_keys
5. Test to make sure you can ssh in without a password. If not,
   play with the permissions some more. Generally speaking, if you
   can't log in without a password, the problem is that either "group"
   or "other" can either read or write some of your files in .ssh

Now that you have ssh working, I'm going to make the assumption that any file that you put at isp.com/home/jblow/public_html/filename is available via http at http://isp.com/~jblow/filename. If that's not the case for you, that's fine. Just adjust the instructions accordingly. Notice how I use environment variables to store the various information I need. One useful thing that many archers do is they'll set some useful environment variables in their loginfile (For example, I use $WORKARCH for james.blackwell@canonical.com--2004, and $BOSSWORK for robert.collins@canonical.com) ::

```
$ export MYARCH=jblow@isp.com--fishfarm-2004
$ export SERVER=sftp://jblow@user.isp.com
$ export ARCHDIR=home/jblow/public_html/archives
$ baz make-archive -l -m $MYARCH $SERVER/$ARCHDIR/$MYARCH
$ baz archive-mirror $MYARCH
```

And that's it. Your archive should now be publicly available. Whenever you want to update the mirror, just run "baz archive-mirror jblow@isp.com--fishfarm-2004".

When you tell other people about your public archive, all that you have to tell them is where it is. In your case, you would just tell them "My archive is at http://user.isp.com/~jblow/archives/jblow@isp.com--fishfarm-2004." Arch will be able to figure out the name on its own.

Probably now is another time for you to take a short break, because we're going to start diving into stuff thats a little more difficult. So why don't you go grab a snack, kiss the spouse, and come right back, ready to go.

5. More Advanced Merging

In the last chapter, we covered how to make your archive publicly available. In this chapter, we're actually going to do something useful with that information. Its time for us to come clean and tell Tux that we've been maintaining a localized version of his software. So, we shoot him an email, and in that email, we tell him something along the lines of:

```
  To: Tux
From: jblow

Hey Tux, just wanted to let you know that your code r0x0rs! Well, it
mostly rocks, cause I had to make a whole bunch of changes. Anyways,
I put my archive up at http://user.isp.com/~jblow/archives
/jblow@isp.com--fishfarm-2004. You can find my hacks in the
fishfarm--jblow--1.2 branch.

Keep up the fight,
Joe Blow
```

Tux is going to do one of two things. He's either going to blow you off for six months (You are, after all, Joe 'Blow'), or he's going to dive into your software with both feet so that he can look productive

to his users. After all, you've already done lots of hard work for him! Lets assume in this case that it's the latter, so that we don't have to wait six months for you to read the next part of the tutorial.

Tux is going to do several things. He's going to register your archive, he's going to merge your work into one of his working copies and, if you're lucky, he's going to commit your work. This would look like this:

```
$ baz register-archive http://user.isp.com/~jblow/archives/
Registered archive: jblow@isp.com--fishfarm-2004
$ baz get fishfarm--devo--1.2 replayingjoe
[tla output]
$ cd replaying
$ baz replay jblow@isp.com--fishfarm-2004/fishfarm--jblow--1.2
[lots of tla output]
$ baz status
[what files changed, etc]
$ baz diffs | less
$ make && make test
[no problems]
$ baz commit -s"Now you can have more than one fishfarm (Joe Blow)"
```

Congratulations! You've just taken your first steps to being a fishfarm developer. Get used to the idea of people yelling at you to work harder, faster, and longer for nothing. :)

However, things aren't quite as simple as they were before. Have you ever tried to apply the same patch twice, and gotten conflicts? Well, the same thing will happen with baz replay, because its not quite smart enough to not apply your own patches. It just so happens that there is a --skip-present option to replay, but that's usually not what we want.

"Why?" I'm Glad you asked. Its possible that when the other guy made a mistake when he merged you by, not just commiting your patches, but his own chan ges as well. If he does that, then your archive is going to think your patch means one thing, but his archive is going to think that your patch means something different. So when you run "baz replay --skip-present", it won't just skip your patch, but it'll skip his local changes as well. Eventually, you're likely to have a conflict, because the your tree looks slightly different to each one of you.

So what should we do? Well, in the next chapters, we're going to cover the merge tool. This tool is a bit smarter, and will let you do things with arch that you just plain can't do with most revision control systems.

6. No muss, No Fuss, Merge away

In the previous chapter, we realized that replay has some limitations that can really cramp the style of developers, particilarly lazy ones that have a habit of comitting local changes along with merges at the same time.

Using the merge tool is essentially as easy as using replay. Tux would do the following:

```
$ baz register-archive http://user.isp.com/~jblow/archives/
Registered archive: jblow@isp.com--fishfarm-2004
$ baz get fishfarm--devo--1.2 mergingjoe
[tla output]
$ cd mergingjoe
$ baz merge jblow@isp.com--fishfarm-2004/fishfarm--jblow--1.2
[lots of tla output]
$ baz status
[what files changed, etc]
$ baz diffs | less
$ make && make test
[no problems]
$ baz commit -s"Now you can have more than one fishfarm (Joe Blow)"
```

So, how does merge work? A quick overly simple explanation is that merge does the following:

1. `baz figures out the last time Tux merged you`
2. `baz builds a list of changes between when he last merged you and what's current in your archive`
3. `baz applies those changes to his working tree`
4. `Tux fixes any rejected code`
5. `Tux commits the merge`

And that's really all there is to it!