Weld Documentation

Release 0.1

Richard Watts, Tony Ibbs

June 26, 2015

Contents

Contents:

CHAPTER 1

Getting started with weld

1.1 As a normal user

If you're a normal user of weld, then there isn't much to learn.

Basically, your project documentation should tell you how to clone a weld - for instance:

```
$ git clone ssh://git@example.com//opt/projects/99/weld project99
```

You can then just develop in `project99` as normal, using git to handle version control as you would for any other project.

1.2 As a muddle user

Your project should include documentation telling you to:

1. Clone a weld
2. `cd` into it and use `muddle init` to set it up

and then you can mostly use the build tree as a normal muddle user.

For instance:

```
$ mkdir project99
$ cd project99
$ git clone ssh://git@example.com//opt/projects/99/weld weld
$ cd weld
$ muddle init weld+ssh://git@example.com//opt/projects/99/weld builds/01.py
```

The `weld+` tells muddle it is dealing with a weld, and (essentially) disables `muddle push` and `muddle pull`. The idea is that you should just use git directly (`git push`, `git pull`, etc.).

1.2.1 A little more detail

A normal muddle build tree looks something like the following:

```
<project>
  .muddle/
  src/
    builds/
```

```
      .git/
      01.py
    base/
      kernel
        .git/
        <lots of source code>
    <and so on>
```

A muddle build tree set up for use with weld instead looks like:

```
<project>
  .git/
  .gitignore
  .muddle/
  .weld/
  src/
    builds/
      01.py
    base/
      kernel
        <lots of source code>
    <and so on>
```

As you can see, there is now a single `.git` directory at the top of the muddle source tree, as well as a `.weld` directory, and a `.gitignore` file. The `.git` directories that would have been present in the `src` directories have gone away - they are not needed in this setup.

The toplevel `.git` directory manages the entire source code tree.

The `.gitignore` file tells git to ignore various things, including the muddle `.muddle`, `obj`, `install` and `deploy` directories.

Use of the `weld+` mechanism in `muddle init` tells muddle *not* to allow `muddle pull` and the like to do anything - the muddle VCS commands are not currently aware of how welds work, and so are disabled by this means. Instead, just use git in the normal manner.

1.3 But where's the use of the `weld` tool?

One of the points of "weld" is that normal users do not need to use the weld command line tool. The idea is that only the software developers maintaining the weld need to worry about how it interacts with its upstream packages. This means that if you're just building software from a weld, it is simply another (albeit perhaps rather large) git repository.

CHAPTER 2

Why weld?

Weld is meant to make it easier to manage the verson control of projects with a moderate to large number of packages. A typical example would be the sources needed to build a Linux system, which might typically contain:

- linux itself
- busybox for the shell and basic command line utilities
- a bootloader
- some kernel modules
- some /etc files
- audio and video support (alsalib, libvorbis, gstreamer, etc.)
- and so on

There are two traditional ways to organise the version control for such a project:

1. One package per repository
2. One repository for the world

2.1 One package per repository

In this approach, each package is put into its own repository (or may, sometimes, be retrieved from the "outside world" repository from which it originates - this has obvious problems if the internet connection to the outside world goes down).

The advantages of this approach are:

- it is very easy to relate the local copy of a package back to its upstream/external version, even if they are not both using the same version control system (e.g., local in git, remote in mercurial or subversion)
- it is easy to keep track of licensing issues, and other such per-package responsibilies, because each package is clearly atomic

On the other hand:

- some form or meta system must be used to decide which packages are required by the particular system that is being built - this is one of the reasons that muddle was first started
- it is hard to make and maintain a coherent change across multiple packages, because there is no linkage at all between the changes in each individual package

- branching across the whole project (for instance, for a release branch) is almost impossible to manage. Muddle provides some help with muddle build trees, but it is still not simple, and not being simple means not being safe/easy to use.
- it is hard to "name" a particular version of the project. Again, muddle provides some support for this with its stamp files, but these are just text files "naming" the repositories and the appropriate commit ids, which is intrinsically clumsy.
- for new software, a decision to split into packages at the wrong granularity (so either too much code in one large package, or too many small packages that are actually tightly integrated) can lead to awkward code management later on
- cloning many small packages is slower than cloning one larger package

2.2 One repository for the world

In this approach, the project as-a-whole has a single repository. Individual packages are imported into this repository, in some appropriate workflow.

> For instance, one might have an import branch for the project, named after its version ("import-busybox-1.2.1"), and once the new version of the package is working, this would be merged back into the main tree.
>
> Alternatively, perhaps, one might have a long-running package specific branch ("package-busybox") into which new versions of the package are periodically copied, tagged with the version number, and then integrated/merged back into the main tree.
>
> The actual mechanism used is not particularly pertinent to this discussion, but we know of people who have good mechanisms in place for handling this sort of repository organisation.

The advantages of this approach are:

- it is very clear what the code being used for the project is - it is that code which is in the repository
- a change can be made across several packages as an individual change
- naming a particular version of the project is as simple as specifying a commit id
- a branch can be made across the whole project - this makes release branching (for instance) manageable

On the other hand:

- it is harder to reason about individual packages when they are all "mashed together" into one place
- it is harder to send changes upstream to the original package repositories when changes to an individual package are not separated out
- if a package is used in two "mega" repositories, but some of the changes (or perhaps just some of the information in commit messages) must not be shared between the two, then moving those changes from one "mega" repository to the original package and thence to the other "mega" repository needs careful management

2.3 Or there's weld

Weld attempts to make it reasonably simple to have something of both worlds.

One VCS is chosen (git) to restrict the complexity of the problem.

weld uses a directory, `.weld`, at the top of your source tree (next to the `.git` directory) to store meta-information about which packages you use and where they come from.

The normal user gets to work in a single large repository, using git, and should mostly be able to ignore the rest of weld.

Setting up a weld in the first place, and relating it to the individual repositories that provide its packages, is regarded as a more expert task, and for this the `weld` commandline tool is provided.

2.3.1 A little more detail

Broadly, the normal user will `git clone` a weld (a single large repository) and work within that as with any other body of source code, using the normal `git` tools to branch/commit/push/pull as required. The intention is that a "normal" user just sees a "mega" repository, and works with it as such.

A project then needs one or more weld managers, who set the weld up in the first place, and curate pushing changes back to individual repositories as and when it becomes necessary, using "weld push" and "weld pull".

2.4 Can I use it with muddle?

Yes, and in fact that was specifically why weld was invented: the multiple repository model traditionally used by muddle makes it quite difficult to track changes. We realised it would be simpler if there was a single git repository for a project which could track back to other repositories.

weld is the tool which makes that possible.

Using weld with muddle is explained in its own section.

2.5 A little terminology: welds, bases and seams

A **weld** is a git repository containing all of the source code for a project.

`weld` is also the command line tool that is used to maintain welds.

A **seam** is a mapping from a directory in an external git repository to the directory in the weld in which it will appear.

Colloquially it is also the directory in the weld that is so described.

A **base** is an external git repository (and implicitly its branch or other specifiers) from which one pulls seams and to which they are pushed.

The term may also be used to refer to the clone of that external directory in the `.weld/bases` directory.

CHAPTER 3

Weld for those who need to maintain welds

3.1 Getting the weld command line tool

Getting weld needs git. If you don't have git on your system, and you're on a Debian based system (Debian, Ubuntu, Linux Mint, etc.), then you can do:

```
$ sudo apt-get install git gitk
```

(the `gitk` program is an invaluable UI for looking at the state of git checkouts - it's always worth checking it out as well as git itself).

Then decide where to put weld. I have a `sw` directory for useful software checkouts, so I would do:

```
$ cd sw
$ git clone https://code.google.com/p/weld/
```

which creates me a directory `~/sw/weld`.

Note: Sometimes (luckily not often) the Google code repositories give errors. In this case, the only real solution is to try again later.

To *use* weld, you can then either:

1. just type `~/sw/weld/weld` - this is the simplest thing to do, but the longest to type.
2. add an alias to your `.bashrc` or equivalent:

```
alias weld="${HOME}/sw/weld/weld"
```

3. add `~/sw/weld` to your PATH:

```
export PATH=${PATH}:${HOME}/sw/weld
```

4. add a link - for instance, if you have `~/bin` on your path, do:

```
$ cd ~/bin
$ ln -s ~/sw/weld/weld .
```

Personally, I use the second option, but all are sensible.

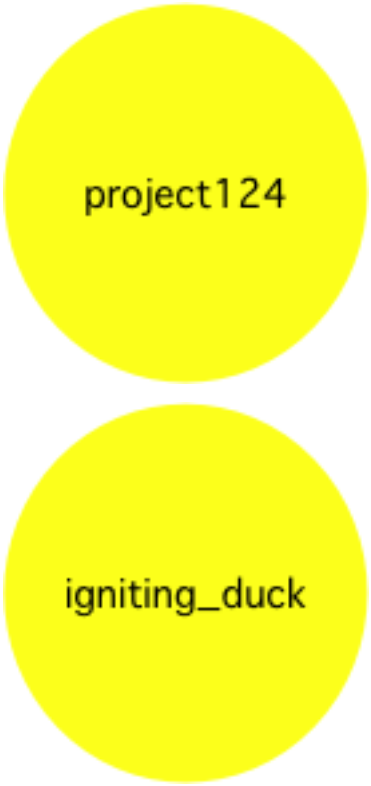
You should now be able to do:

```
$ weld help
```

and get meaningful output.

3.2 Creating a weld

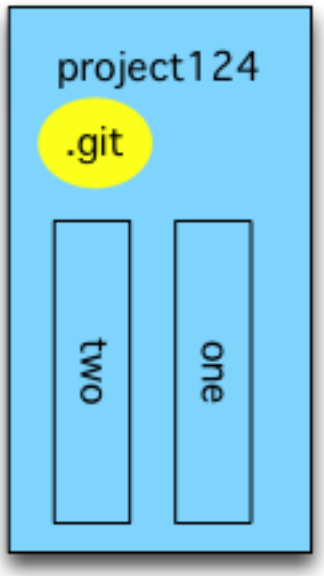
We start with two external packages, project124 and igniting_duck. Here are their remote repositories:

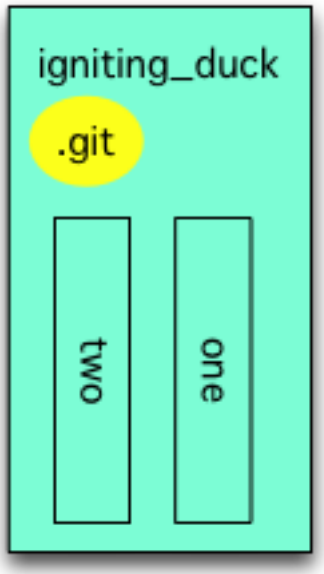


We can clone them in the normal manner:

```
$ git clone file://<repo_base>/project124
$ git clone file://<repo_base>/igniting_duck
```

which gives us two working directories:





Each has directories called `one` and `two`, and naturally each has its own `.git` directory.

To create a new weld, we must first write an XML file describing it. For instance, we might create a file called `frank.xml` containing the following:

```
<?xml version="1.0" ?>
<weld name="frank">
  <origin uri="file://<repo_base>/fromble" />
  <base name="project124" uri="file://<repo_base>/project124"/>
    <seam base="project124" dest="124" />
  <base name="igniting_duck" uri="file://<repo_base>/igniting_duck" />
    <seam base="igniting_duck" source="one" dest="one_duck" />
    <seam base="igniting_duck" source="two" dest="two_duck" />
</weld>
```

This says that:

1. The name of the weld is `frank`
2. The remote repository for the weld is (will be) `file://<repo_base>/fromble`
3. The weld will contain two bases:
   (a) The first base is called `project124`, and its remote repository is `file://<repo_base>/project124` - in other words, it's the first of the two repositories we have already been introduced to at the start of this chapter.

   Note that the name we give the base does not have to match the repository name (although it probably normally will).

   (b) The second base is called `igniting_duck`, and it is the second remote repository from above.
4. The `project124` base will provide a single seam in the weld. The source directory is not specified, so this will be the entire content of the base. The seam will be put into the weld as directory `124`
5. The `igniting_duck` base will provide two seams in the weld. The base directory called `one` will be stored in the weld as `one-duck`, and the base directory called `two` will be stored in the weld as `two-duck`. Any other directories in the base will not be added to the weld.

Note: Why use XML?

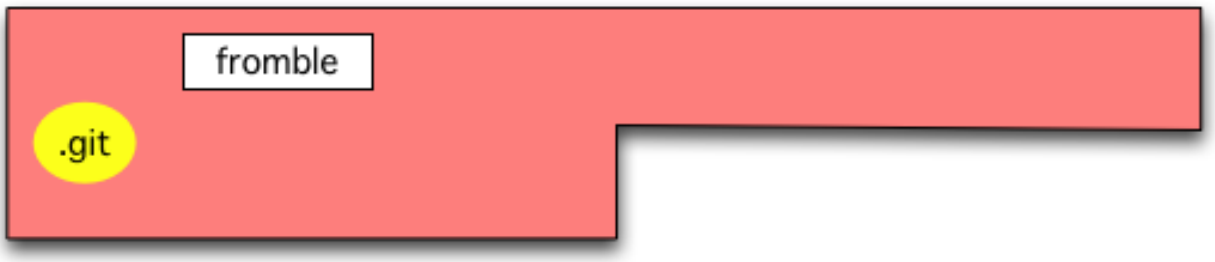
We did consider using Python, but felt that given the highly declarative nature of the information described, the number of opportunities for self-mutilation was just too high.

And because all the best shell scripts parse XML, we chose that.

Once we've got the XML file, we can use `weld init`:

```
$ mkdir fromble
$ cd fromble
$ weld init ../frank.xml
> git init
> git add fromble/.weld/welded.xml .gitignore
> git remote rm origin
> git remote add origin file://<repo_base>/fromble
> git commit --allow-empty --file /tmp/weldcommitYp7JZ2
Weld initialised OK.
```

to create an empty weld:



The weld contains:

```
fromble/
  .git/...
  .gitignore
  .weld/
    welded.xml
```

(I've left out the content of the `.git` directory).

The `.gitignore` instructs git to ignore some artefacts that weld knows it will create in the `.weld` directory. The `.weld/welded.xml` is a "copy" of the original `frank.xml` (actually, it is produced by reading the original XML and then writing it out from the internal datastructure, so the layout is likely to be slightly different, and any comments will be lost, but the content should have the same effect). Both have been committed to the weld's git repository.

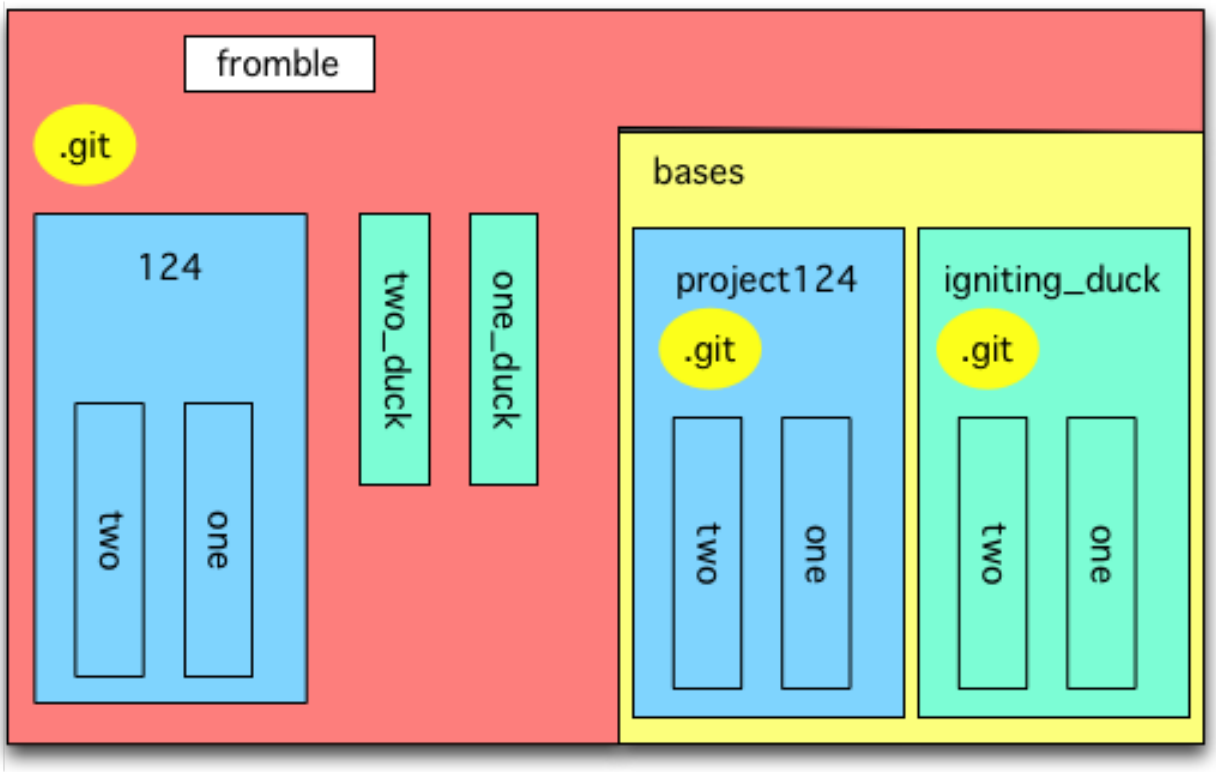
To populate our weld, we need to use:

```
$ weld pull _all
```

This clones the two remote repositories into the weld's `.weld/builds` directory:

```
.weld/bases/project124
.weld/bases/igniting-duck
```

then copies the content of those clones into the appropriate places in the weld, and commits the new weld contents. This gives us:



Our directory structure now looks like:

```
fromble/
  .git/...
  .gitignore
  .weld/
    bases/
      igniting_duck/
        .git/...
        one/
          <source-code ign-1>
        two/
          <source-code ign-2>
      project124/
        .git/...
        one/
```

```
          <source-code 124-1>
        two/
          <source-code 124-2>
    counter
    welded.xml
  124/
    one/
      <source-code 124-1>
    two/
      <source-code 124-2>
  one-duck/
    <source-code ign-1>
  two-duck/
    <source-code ign-2>
```

Here we can see that in `.weld/bases` are the clones of the two remote packages (our "bases"), each with its own `.git` directory. A normal user will never interact with these, and strictly speaking they are not part of the weld.

We can also see, at the top level of `fromble`, that we now have three source directories: `124`, `one-duck` and `two-duck`. These are checked into the weld's git repository, and correspond to the seams described in the XML file. Thus the weld source directory `124` corresponds to all of the `project124` base, whilst the two directories in the `igniting_duck` base have been split into separate (in this case top-level) directories in the weld, just as the XML file described.

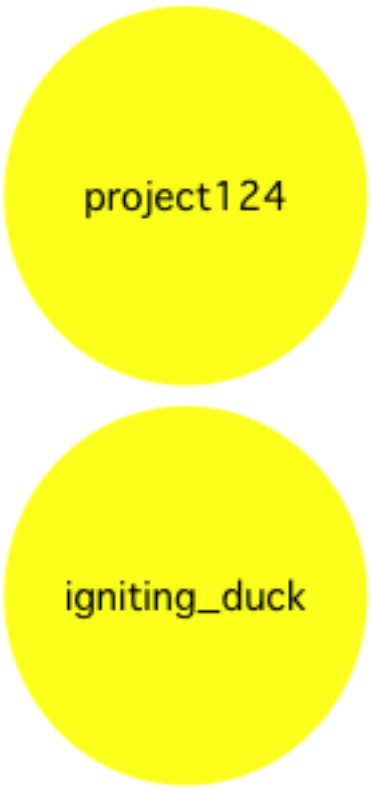
Now we've got our weld set up, we can create a bare repository for it in the normal manner - in this case:

```
$ pushd <repo-base>
$ mkdir fromble
$ cd fromble
$ git init --bare
$ popd
```

and push to it (`weld init` set up the URI in the XML file as the origin remote, so this "should just work"):

```
$ git push master origin
```

so we now have three remote repositories:

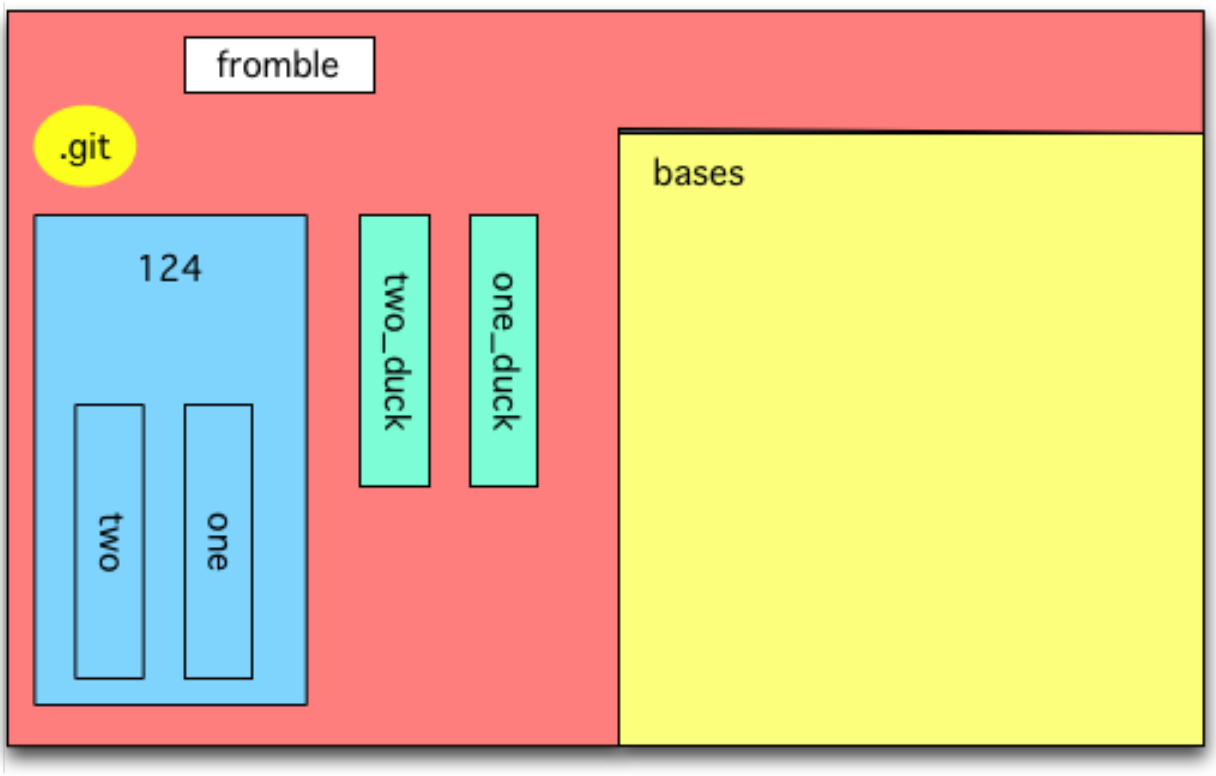


3.3 Using the weld just needs git

Another user can now clone the weld directly:

```
$ cd ~/work
$ git clone file://<repo_base>/fromble
```

which gives them the weld with its seams:



This new weld has the following directory structure:

```
fromble/
  .git/...
  .gitignore
  .weld/
    counter
    welded.xml
  124/
    one/
      <source-code 124-1>
    two/
      <source-code 124-2>
  one-duck/
    <source-code ign-1>
  two-duck/
    <source-code ign-2>
```

Note: In normal use of a weld, there is no `.weld/bases` directory. The bases are not part of the weld itself, they will only be retrieved if the user runs a weld command that needs them.

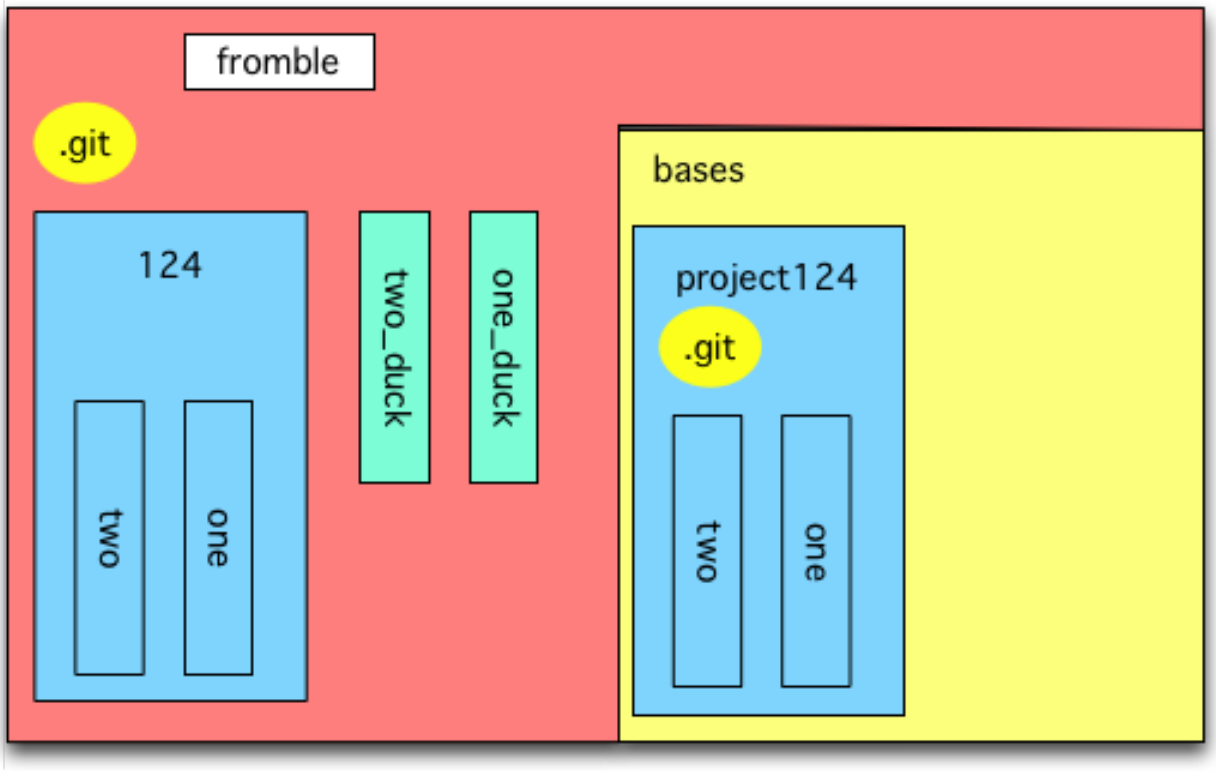
The user can work on the content of the weld as they need, pushing to and pulling from the weld's remote repository with git in the normal manner.

3.4 Using `weld` commands on the weld may pull bases

The `weld` command line tool will download (clone or update) the bases when it needs to. For instance, some queries need access to the base. In particular:

```
$ weld query base project124
```

will clone project124 into `.weld/bases/`, giving us:



or:

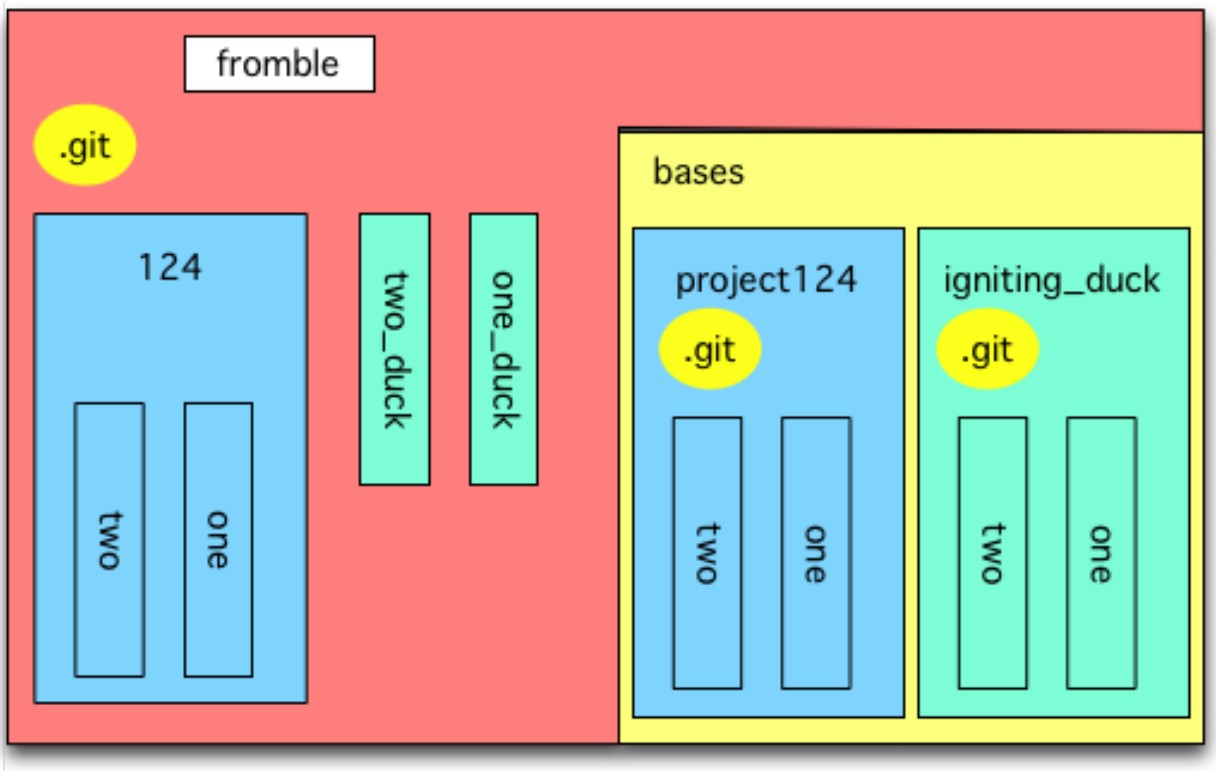
```
fromble/
  .git/...
  .gitignore
  .weld/
    bases/
      project124/
        .git/...
        one/
          <source-code 124-1'>
        two/
          <source-code 124-2'>
    counter
    welded.xml
  124/
    one/
      <source-code 124-1>
    two/
      <source-code 124-2>
  one-duck/
    <source-code ign-1>
  two-duck/
    <source-code ign-2>
```

Note that the source code in the project124 base may be different than that in the corresponding seams (shown here as `<source-code 124-1'>` versus `<source-code 124>`) - which is exactly what the query needs to know.

If we decided to update the weld with any changes made in the remote igniting_duck repository:

```
$ weld pull igniting_duck
```

then this would also necessitate pulling the base:



In this case, the source code in the weld would be updated to match that in the igniting_duck base:

```
fromble/
  .git/...
  .gitignore
  .weld/
    bases/
      igniting_duck/
        .git/...
        one/
          <source-code ign-1'>
        two/
          <source-code ign-2'>
      project124/
        .git/...
        one/
          <source-code 124-1'>
        two/
          <source-code 124-2'>
    counter
    welded.xml
  124/
    one/
      <source-code 124-1>
    two/
      <source-code 124-2>
```

```
  one-duck/
    <source-code ign-1'>
  two-duck/
    <source-code ign-2'>
```

3.5 Adding a base

Briefly:

1. Make sure that the base to be added already exists as a remote git repository.
3. In the weld, do a `git pull` to make sure that the weld is up-to-date.
3. Edit the `.weld/welded.xml` file to add the base (including its repository URI) and the seam(s) you want from that base.

   Commit the XML file with `git commit .weld/welded.xml` and an appropriate message.
4. Run `weld pull <new-base-name>` to pull the base.

   This will:

   (a) Clone the base repository into `.weld/bases/<new-base-name>`.

   (b) Copy the code for the seam(s) selected into the weld.

   (c) Commit the results.
5. Once you are happy that the base and seam(s) are integrated properly into the weld, then use `git push` to push the weld to *its* remote repository.

3.6 Changing a base or seam

At the moment, altering the content of the weld, as described by the XML file, needs some care.

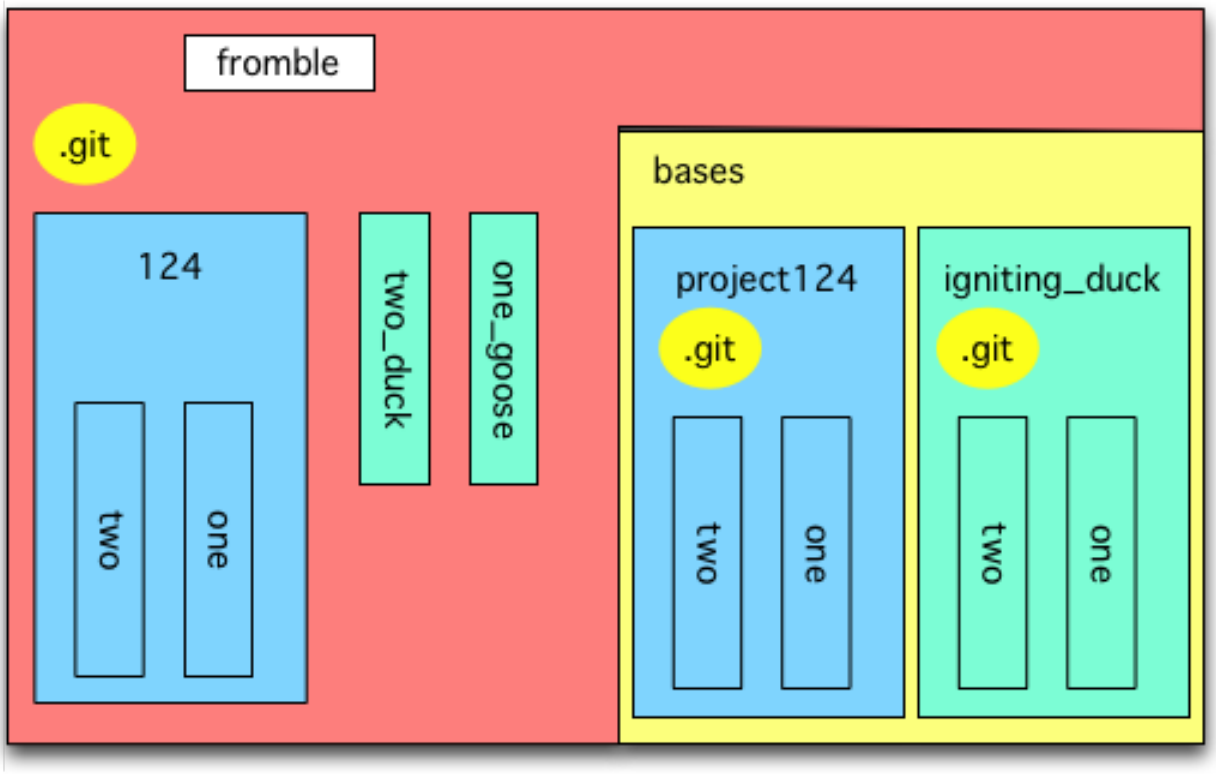
For instance, if we edited the XML file to change name of the seam `one_duck` to `one_goose`:

```
<?xml version="1.0" ?>
<weld name="frank">
  <origin uri="file://<repo_base>/fromble" />
  <base name="project124" uri="file://<repo_base>/project124"/>
    <seam base="project124" dest="124" />
  <base name="igniting_duck" uri="file://<repo_base>/igniting_duck" />
    <seam base="igniting_duck" source="one" dest="one_goose" />
    <seam base="igniting_duck" source="two" dest="two_duck" />
</weld>
```

and did:

```
$ git mv one_duck one_goose
```

(one day, weld may provide a command to do those together for you) then we would have:



and because we have done *both* of those things, `weld pull`, `weld push` and all the other weld commands would recognise that the base directory `igniting_duck/one` is now related to the weld seam `one_goose`.

3.7 Things to remember not to do in a world of welds

Do not use git submodules in bases, as weld will not preserve them.

Do not use commit messages that start "X-WeldState:", as weld uses such for its own purposes.

Do not use branches that start "weld-", as weld uses such for its own purposes (and is not very careful in checking if you're on an offending branch, it just looks to see if the name starts with "weld-").

Do not change the name of the `origin` remote of a weld, as the weld command assumes that `origin` is the origin remote it should use.

CHAPTER 4

Other information

This chapter is being rewriten, and its parts moved around/elsewhere. Some of it may be inaccurate. Please be patient.

4.1 The weld XML file

A simple weld XML file:

```
<?xml version="1.0" ?>
<weld name="frank">
  <origin uri="ssh://git@home.example.com/ribbit/fromble" />
  <base name="project124" uri="ssh://git@foo.example.com/my/base" branch="b" rev=".." tag=".."/>
  <base name="igniting_duck" uri="ssh://git@bar.example.com/wobble" />
  <seam base="project124" dest="flibble" />
  <seam base="igniting_duck" source="foo" dest="bar" />
</weld>
```

This file tells weld:

- This weld is called frank. This name is not used for anything at the moment (caveat: It is put into the "X-Weld-State: Pushed" markers in the base, but otherwise never referenced).
- The origin for this weld is at ssh://git@home.example.com/ribbit/fromble.
- This weld draws from two bases: project124 and igniting_duck.
- project124 turns up in a directory in the weld called flibble.
- igniting_duck/foo turns up in <weld>/bar

4.1.1 Details

The XML file must:

- start with an XML version line: `<?xml version="1.0" ?>`, because otherwise it wouldn't be XML
- continue with the start of a weld definition: `<weld name=`*name*`>`. Whilst the weld name is required, it is not currently used for anything.
- which contains an `<origin uri=`*origin*`/>` entity (althogh the `uri` is optional at the moment)
- as well as zero or more base and seam definitions
- and end with the end of a weld definition: `</weld>`

Comments are allowed as normal, and are ignored (and will not be retained when the XML file is copied).

Only one weld definition is allowed in a file, although this may or may not be checked - regardless, any weld definitions after the first are ignored.

With a weld definition, there are two types of entity:

- base definitions
- seam definitions

A base definition must occur before the seam definitions that belong to it, but otherwise the order of entries is not important. A seam may only belong to a single base. A base without any seams is not forbidden by the file syntax, but will not be of much use.

A base definition contains:

- `name` - the name of this base
- `uri` - the URI for the repository from which this base is to cloned/checked out
- `branch`, `rev` or `tag` - the branch, revision or tag to clone/check out. These defaut to "master", "HEAD" and (essentially) "HEAD" respectively. It is not currently defined what happens if you specify more than one of these for a particular base.

A seam definition contains:

- `base` - the name of the base that this seam belongs to. This base must already have been defined in the XML file.
- `name` - the name of this seam. This is optional, and defaults to None
- `source` - the directory in the bare repository from which the seam's contents are taken. This is optional, and defaults to `"."`, meaning the top (root) of the repository. The `source` may specify a sub-directory, e.g., "src/base/libuseful".
- `dest` - the directory in the weld to which the seam's contents should be copied. This is optional and defaults to `"."`, meaning the top (root) of the weld. The `dest` may specify a sub-directory, e.g., "kynesim/main/useful".
- `current` - This is optional and is not used at the moment - it may be withdrawn in future versions of weld.

It is not defined what happens if the same base or seam is defined more than once (with either the same values or different values). Future versions of weld may reject an XML file that does this.

It *is* intended that two bases with diffferent names be regarded as different, although what happens if that is the only difference between them is not defined.

Warning: *Do not cross the streams.*
Specifically, no two different seams should have the same destination, lest weld get terribly confused. `"."` counts, so you cannot have more than one seam with no `dest`. This also means that destinations that "nest" (e.g., `src/fred` and `src/fred/jim`) are forbidden.

Note: I am not aware of anything that ensures that the origin URI corresponds to the place that you actually clone the weld from. Indeed, since it is a URI (and not a URL), it need not so correspond. However, if you do something obscure based on this, then no-one is going to like you.

4.2 Files in .weld

When you first clone a weld, the only file in the `.weld` directory will be:

- `welded.xml` - this is the file that describes this weld. It is a copy of the XML file given to weld init.

After doing a `weld pull`, a `weld push`, or a `weld query` on a base (which may need to "pull" the base to find out about it), there will also be:

- `bases/` - this is a directory containing a clone of each of your bases, retrieved as they are needed.

You may also see:

- `counter` - this is a file whose content counts upward. It is used to force changes so we never have empty commits when doing `weld pull` (or `weld push`). It appears to be necessary because - `git commit --allow-empty` can sometimes lose commits.

During a `weld pull` or `weld push` you will also see:

- `complete.py` - the script that `weld finish` runs.
- `abort.py` - the script that `weld abort` runs.

and `weld push` also creates:

- `pushing/` - which contains the commit message to be used at the end of the `weld push`, and a marker to indicate whether a merge is in progress.

All of these should be deleted when the `weld pull` or `weld push` is finished (and, specifically, `weld finish` and `weld abort` should delete them).

4.3 A summary of weld commands

weld init <weld-xml-file>

> This command takes a <weld-xml-file> that you have written and creates a git repository for it.
>
> The XML file is written to `.weld/welded.xml`.
>
> An initial `.gitignore` file is created, which tells git to ignore various weld working files, including `.weld/bases`.

weld pull <base-name>

> The special "name" `_all` means "pull all bases".

weld finish

> Finish a weld pull or push that had problems (indicating that the problems were fixed).

weld abort

> Abort a weld pull or push that had problems (thus discarding it)

weld query bases

> List the bases, and their seams

weld query base <base-name>

> Report on the current state of the named base.

weld query seam-changes <base-name>

> Report on the seam changes for the named base.

weld status

If we are part way through a `weld pull` or `weld push` say so.

Otherwise, report on whether we should do a `git pull` or `git push` of our weld. This is intended to be useful before doing a `weld pull` or `weld push` of our bases.

4.4 Commit messages that weld inserts

Weld will occasionally leave commits containing messages to itself. It is important that you do not start any other commit messages with `X-Weld-State`

The messages it leaves are:

X-Weld-State: Init

Indicates that the weld started here (with nothing merged)

X-Weld-State: PortedCommit <base-name>/<commit-id> [<seams>]

Indicates that it ...

X-Weld-State: Seam-Added <base-name>/<commit-id> [<seams>]

Indicates that it ...

X-Weld-State: Seam-Deleted <base-name>/<commit-id> [<seams>]

Indicates that it ...

X-Weld-State: Seam-Changed <base-name>/<commit-id> [<seams>]

Indicates that it ...

X-Weld-State: Merged <base-name>/<commit-id> [<seams>]

Indicates that it merged <base-name> <commit-id> with the following seams.

X-Weld-State: Pushed <base-name>/<commit-id> [<seams>]

Indicates that it ...

Note that the `X-Weld-State: Seam-` messages only occur in the branches on which base merging is done.

In the base repositories, it can also leave a commit message of the form:

```
X-Weld-State: Pushed <base-name> from weld <weld-name>
```

This commit will then contain a sequence of lines, each of which is (currently) the "short" SHA1 id for a squashed component commit, followed by its one line summary - so for instance:

```
X-Weld-State: Pushed igniting_duck from weld fromble

e8addb1 Add trailing comments across the bases and to the weld
7eaa68a One-duck: Also build one-duck, same as one
f589384 One-duck: Add a comment to the end of the Makefile
```

The format of this message may change in the future.

CHAPTER 5

Creating a muddle build tree for use with weld

Setting up a muddle build tree for use as a weld is still to be documented.

Note: Steps will include:

1. Create your muddle build description, with appropriate repository information
2. Commit it to a remote/bare repository, as one does
3. Use muddle to find out the list of packages and their repositorie
4. Use that information to write the weld XML file
5. Follow the normal instructions on creating a weld given its XML file

Weld was written with the intent of being muddle-compatible.

A normal muddle build tree looks something like the following:

```
<project>
  .muddle/
  src/
    builds/
      .git/
      01.py
    base/
      kernel
        .git/
        <lots of source code>
    <and so on>
```

A muddle build tree set up for use with weld instead looks like:

```
<project>
  .git/
  .gitignore
  .muddle/
  .weld/
  src/
    builds/
      01.py
    base/
      kernel
        <lots of source code>
    <and so on>
```

The toplevel `.git` directory manages the entire source code tree.

The `.gitignore` file tells git to ignore various things, including the muddle `.muddle`, `obj`, `install` and `deploy` directories.

Typically, a user just needs to do something like:

```
$ git clone ssh://git@example.com//opt/projects/99/weld weld
$ cd weld
$ muddle init weld+ssh://git@example.com//opt/projects/99/weld builds/01.py
```

after which they can do `muddle build _all` and so on in the traditional manner.

Use of the `weld+` mechanism in `muddle init` tells muddle *not* to allow `muddle pull` and the like to do anything - the muddle VCS commands are not currently aware of how welds work, and so are disabled by this means. Instead, just use git in the normal manner.

Note: It is possible that muddle may become more "weld aware" in the future, but to be honest the current mechanism seems like a sensible first approach, and may be the correct way to handle this in the long term as well.

CHAPTER 6

How weld pull and weld push work

6.1 How "weld pull" does its stuff

Obviously, the code is the final word on what happens, but this is intended as reasonable background.

Remember, "weld pull" updates its idea of the bases, and then updates the seams in the weld "to match".

> The main code for this is in `welded/pull.py`

6.1.1 The very short form

We make sure we have a current copy of the base (in `.weld/bases`), copy across the changes for each seam in that base to our weld, and commit with an `X-Weld-State: Merged` message.

- The local copy of the base is updated, but is not otherwise changed.
- The relevant seams in the weld are updated to match the equivalent directories in the base.

6.1.2 The short form

- Check it's safe to do a `weld pull`
- Update the copy of the base in `.weld/bases`
- Find the last time `weld pull` or `weld push` was done for this base - this is the synchronisation point.
- Determine which seams have been added/removed/changed in comparison to the newly update base. If nothing has changed, then there is nothing to do.
- If there are deleted seams, delete them in the weld. If there are added seams, add them in the weld, by copying their files across. If there are changed seams, then replay the appropriate changes in the weld.
- Commit with an `X-Weld-State: Merged` message.

6.1.3 In detail

So we're pulling a base

> (You can also pull multiple bases at once, by giving multiple base names on the command line, or use `weld pull _all` to pull all bases, but these both just work by doing this whole sequence for each base in turn. Note that this can be more confusing, for instance if the Nth base requires remedial action

to take to "finish" it, at which point you have to fix the problem, do `weld finish`, and then give the original weld command again to pull the remaining bases.)

(Pulling an individual seam would in theory be possible, but rather fiddly, and of questionable use anyway, so we'll go with just pulling bases).

Given the name of a base:

1. Weld checks that there are no local changes in the weld - specifically, it runs `git status` in the weld's base directory (the directory containing the `.weld` directory). If there are any files in the weld that could be added with `git add` or committed with `git commit`, then it will refuse to proceed, suggesting that the user commit or stash the changes first.

   It also checks whether:

   - the user is part way through an unfinished `weld pull` or `weld push`
   - the weld could be updated with `git pull` (it looks at the remote repository to determine this)
   - the current branch is a weld-specific branch (starting with `weld-`).

   and refuses to proceeed if any of those is true (this is essentially what `weld status` does, so you can do it beforehand as well).

2. It finds the last time that `weld pull` or `weld push` was done for this base, by looking for the most recent commit with an `X-Weld-State: Merged <base-name>` or `X-Weld-State: Pushed <base-name>` message. If there isn't one (i.e., this is the first `weld pull` or `weld push` for this weld), then it uses the `X-Weld-State: Init` commit instead.

   For `weld pull` we want to know the last time our base was synchronised with the weld as a whole. Since both `weld pull` and `weld push` do this, we can use either as the relevant place to work from.

3. Weld makes sure its copy of the base is up-to-date:

   (a) If it doesn't yet have a clone of the base, it does:

```
$ cd .weld/bases
$ git clone <base-repository> <base>
$ cd <base>
$ git pull
```

   (b) If it does have a clone of the base, it does:

```
$ cd .weld/bases/<base>
$ git pull
```

   In either case, it notes the HEAD commit of the base.

4. It determines which (if any) seams have been deleted, changed or added in the weld (with respect to the now up-to-date base). If all of those lists are empty, there is nothing to do, and the `weld pull` for this base is finished.

5. It branches the weld. The branch point is the synchronisation commit that was found earlier (the last `weld pull` or `weld push` commit, or else the Init commit).

   (The branch name used is chosen to be unique to this repository, and is currently of the form "weld-merge-<commit-id>-<index>", where <commit-id> is the first 10 characters of the synchronisation commits SHA1 id, and <index> is chosen to make the branch unique in case that is not enough.)

6. It then:

   - deletes any deleted seams

- modifies any modified seams
- adds any added seams

within that branch.

Deleting a seam is easy - it just means deleting the appropriate directory.

Adding a seam just copies the directory structure for that seam across from the base into the correct place in the weld.

Modifying a seam uses `git diff` to determine the appropriate changes in the base, and then replays them in the weld.

Note: *TODO* It occurs to me that the technique use in `weld push` might be more efficient than this last, if it turns out to be usable - I'd need to think on this further. (Tibs)

---

7. It writes `.weld/complete.py` and `.weld/abort.py`, which can later be used by the `weld finish` and `weld abort` commands if necessary (and which will be deleted if the `weld pull` of this base doesn't need user interaction).
8. It merges the original branch (typically `master`) onto this temporary branch. This will commonly "just work", but if anything goes wrong, the `weld pull` stops with a return code of 1 and a message of the form:

```
<merge error message>
Merge failed
Either fix your merges and then do 'weld finish',
or do 'weld abort' to give up.
```

9. If the merge onto the branch succeeded, or if the user fixes problems and then does `weld finish`, then the `complete.py` script is run, which:
   (a) changes back to the original branch
   (b) calculates the difference between this branch and the temporary branch on which we did our merge
   (c) applies that patch to this original branch
   (d) makes sure that any changed files are added to the git index (it does this over the entire weld, but that should be OK because nothing else should be changing the weld whilst we're busy)
   (e) commits this whole operation using an appropriate `X-Weld-State: Merged <base-name>` message.
   (f) deletes the `complete.py` and `abort.py` scripts

   At the moment, this doesn't delete the temporary/working branch (which will show as a loop if you look in gitk). Future versions of weld may do so as part of the "complete" phase, but during the current active development it's thought to be useful to leave the branch visible.
10. If the merge didn't succeed, and the user chooses to do `weld abort`, then the `abort.py` script is run, which:
    - switches back to the original branch
    - deletes the temporary/working branch
    - deletes the `complete.py` and `abort.py` scripts

Also note that the `weld-` branches are always meant to be local to the current repository - they're not meant to be pushed anywhere else.

6.2 Not having those "remotes/origin/weld-" branches

If you do a `weld pull` and then do a `git push` of the weld, in general the transient branches will not be propagated to the weld's remote.

However, if you clone directly from a "checked out" weld (rather than from a bare repository), then by default all branches are cloned, which is (a) untidy, and (b) mak cause future working branches to have the same name as earlier (remote) working branches.

If you have git version 1.7.10 or later, then you can instead clone a "working" weld using:

```
$ git clone --single-branch <weld-directory>
```

to retrieve (in this case) just `master` (or use `-b <branch` to name a specific branch).

Of course, unfortunately, if you later do a `git pull`, then the branches will be fetched for you at that stage, so it's not a perfect solution. But then maybe you shouldn't clone a "checked out" weld.

6.3 How "weld push" works

Obviously, the code is the final word on what happens, but this is intended as reasonable background.

Again, we're only going to look at doing "weld push" on a single base - the command line will take more than one base name, or the magic `_all`, but we'll ignore that here.

> The main code for this is in `welded/push.py`

6.3.1 The very short form

We make sure we have a current copy of the base (in `.weld/bases`), copy across the changes for each seam in that base from our weld to the base, commit them all as one change with an `X-Weld-State: Pushed` message, and push to the base's origin. We also add an empty `X-Weld-State: Pushed` commit in the weld, as a marker of when the `weld push` happened.

- The base is updated to match its seams in the weld, and pushed to its remote.
- The weld is marked with when the push happened.

6.3.2 The short form

- Check it's safe to do a `weld push`
- Update the copy of the base in `.weld/bases`
- Determine the last `weld push` for this base
- For each seam, work out which files have changed (added, removed, changed) in the weld, since that last `weld push`
- Use `rsync` to make the files in (the corresponding directory in) the base match
- Commit that in the base, and push it to the base's remote
- Add a corresponding `X-Weld-State: Pushed` commit in the weld

Remember that only seams that are currently "named" in the weld are pushed, since they're the only seams that are of interest "now" - if the user wanted to push changes to a seam that is not currently in use, then they should have done it when it was in use.

6.3.3 In detail

So doing `weld push` for a given base name works as follows:

1. Weld checks that there are no local changes in the weld - specifically, it runs `git status` in the weld's base directory (the directory containing the `.weld` directory). If there are any files in the weld that could be added with `git add` or committed with `git commit`, then it will refuse to proceed, suggesting that the user commit or stash the changes first.

   It also checks whether:

   - the user is part way through an unfinished `weld pull` or `weld push`
   - the weld could be updated with `git pull` (it looks at the remote repository to determine this)
   - the current branch is a weld-specific branch (starting with `weld-`).

   and refuses to proceeed if any of those is true (this is essentially what `weld status` does, so you can do it beforehand as well).

2. Weld makes sure its copy of the base is up-to-date:

   (a) If it doesn't yet have a clone of the base, it does:

```
$ cd .weld/bases
$ git clone <base-repository> <base>
$ cd <base>
$ git pull
```

   (b) If it does have a clone of the base, it does:

```
$ cd .weld/bases/<base>
$ git pull
```

3. It finds the last time that `weld push` was done for this base, by looking for the most recent commit with an `X-Weld-State: Pushed <base-name` message. If there isn't one (i.e., this is the first `weld push` for this weld), then it uses the `X-Weld-State: Init` commit instead.

   Why do we use the last `weld push`, and not the last `weld push` *or* `weld pull`?

   Consider the following "pseudo git log":

```
6   +   change B to a file in base <fred>
5   o   X-Weld-State: Merged <fred>/...
4   -   a change to some irrelevant file(s)
3   +   change A to a file in base <fred>
2   o   X-Weld-State: Pushed <fred>/...
1   x   some common commit
```

   So we last did `weld pull` at commit 5, and the weld thus contains all the changes from base <fred> up to that point.

   However, we last did a `weld push` at commit 2, which means that changes 3 and 6 have still to be applied to base <fred>. But change 3 is before our last `weld pull`, so we definitely want the last `push`.

   Note: Remember: `weld pull` updates the base from its remote, and then brings any changes therein into our weld. It does not propagate any changes in the weld back to the base.

   ---

4. Weld looks up the current seams being used for this base. This tells it which directories (in the weld and in the base) it is interested in.

5. It looks up all of the changes in the weld since the synchronisation point (using `git log --one-line`) and remembers them.

   If there aren't any, it has finished.

6. It trims out any `X-Weld-State` commits from that list, and remembers *it*.

   Again, if there are no changes (left), then it has finished.

7. As an aid during development (so this may go away later on), it tags the synchronisation commit, using a tag name of the form `weld-last-<base-name>-sync-<commit-id>`, where <commit-id> is the first 10 characters of its SHA1 commit id.

8. In the base, it branches at the synchronisation point (remember, `X-Weld-State` commit messages record the equivalent commit id in the base as well), using a branch name of the form "weld-pushing-<commit-id>-<index>".

9. We then update the branch in the base:

   For each seam that the base currently has in our weld:

   (a) We use `git ls-files` in the appropriate seam in the weld to find out which files git is managing.

   (b) We do the same in the corresponding directory in the base.

   (c) For files which the seam (in the weld) has, but the base does not, we do `git rm`. We commit that change.

   (d) For all the other files in the seam, we just copy them over into the base (actually, we use `rsync`). It is, of course, quite likely that many of them won't have changed, but that's OK. Then we `git add` all of the files we've copied, in the base, and commit that change as well.

   Note: Any seams that are not in the weld are, by definition, not of interest to us. Even if they were in the weld at the last synchronisation point, the fact they aren't *now* means we are not interested in any (possible) intermediate changes - if we cared about such changes, we should have done `weld push` then.

   ---

10. We prepare a final commit message, and write out the `.weld/complete.py` and `.weld/abort.py` files.

11. We run the `complete.py` file to finish off our `weld push`. This:

    - sets the merging indicator (touches a file in the `.weld/pushing` directory)
    - in the base, if the merging indicator is not set, merges the original branch (normally "master") into our working branch - this should just proceed with no problems
    - still in the base, merges that back into the original branch
    - still in the base, commits the change using the saved commit message

      The commit message has a summary of the corresponding commits from the weld, as output by `git log --one-line`.

      If the user specified `weld push -edit`, then they get the chance to edit the message before it is used.

    - notes the new HEAD commit id in the base
    - adds an `X-Weld-State: Pushed <base-name>/<commit-id>` commit to the weld, using that commit id (this is, of course, notionally an empty commit).
    - deletes the merging indicator and the saved commit message.

If something did go wrong, then `weld finish` just does that last item again (which is why we need the merging indicator). `weld abort` deletes the working branch, and then deletes the merging indicator and saved commit message.

CHAPTER 7

To do list

The following is the current known "to do" list. This is above and beyond any list of issues on the google code site.

7.1 Branch, tag, commit support for seams

Priority: high

The weld XML file allows specifying a branch, tag or revision (commit id?) for a seam.

- It is not defined what happens if you specify more than one. Ideally this would not be allowed, and the user would be told so.
- I'm not sure what, if any, of the code takes notice of these. `push` certainly doesn't. This needs fixing.

This is the most important thing to fix, and must be thoroughly tested.

7.2 XML file format

Priority: medium

The current XML file is laid out as:

```
<base name=A ... />
<seam base=A ... />
```

and so on, where the seams for a base must occur after the base entity.

Would we be better with a "more traditional" layout like:

```
<base name=A ... >
    <seam ... />
</base>
```

which removes the need for the implicit ordering.

If we go for this change, we should add a "version" attribute to the <weld> entity, and make this `<weld name=XXX version=2>` so that we can continue to support old format files.

> (We *could* actually detect both styles of file by inspection, and support them that way, even allowing mixed format (!) files, but using a version number feels cleaner.)

7.3 Base and seam commands

Priority: medium

Adding, deleting and renaming seams (and bases) is fiddly. We should probably provide some commands to bundle up all the actions necessary. For instance:

- `weld add base <base_name> <uri>` - adds the base to the XML file, clones it into the "bases" directory (so the user can inspect it to figure out seam names).
- `weld delete base <base_name>` - removes the base and all its seams from the XML file, deletes its clone from the "bases" directory if it is there, deletess all the seams from the weld.
- `weld add seam <base_name> <seam_name> <from-dir> <to-dir>` - add the seam to the XML file (the base must already exist), and set up the seam. Doesn't do a `git commit` - that's up to the user.
- `weld delete seam <base_name> <seam_name>` - removes the seam from the XML file, and removes its directory from the weld. Again, doesn't do a `git commit`.
- `weld rename seam <base_name> <old_seam_name> <new_seam_name>`

Since creating the initial weld means writing an XML file, maybe we should also provide `weld init --empty <weld-name> <uri>` to create a weld with a minimal XML file - the above `weld add` commands can then be used to populate it with something more interesting.

Technically, we can manage with just those, but it's probably also friendly to provide:

- `weld move seam <base_name> <seam_name> <old_to_dir> <new_to_dir>`
- `weld rename seam <base_name> <old_seam_name> <new_seam_name>`

and maybe some others as becomes evident with time.

7.4 Weld origin URI

Priority: medium

Should we be checking this against the *actual* origin that we are using for the weld (i.e., check the origin URI declared in the XML file against the origin that git is using)?

What commands should check it, and what should they do if the values disagree?

7.5 Weld pull and push common code

Priority: medium

There is some common code between weld push and weld pull (pylint notices a small amount of it).

Moreover, it is possible that the "use `git ls-files to find files and then copy them` approach used by weld push might be applicable in weld pull as well.

Furthermore, weld push uses a `pushing/` directory to keep its temporary files local - again, weld pull could do the same with a `pulling/` directory. This has the advantage that a partially completed push or pull is resumable over a system reboot (when the `/tmp` files would be deleted).

7.6 Command line

Priority: low

- `weld -h` could be more informative
- `weld help` should be paged (as with muddle and git), and the formatting could be better.
- `weld help <command>` should be implemented
- Command line switches that only apply to one or two commands should not be general (I'm thinking of `-tuple` and `-edit` particularly)

7.7 Weld push commit message content

Priority: low

The commit message (in the base) from `weld push` takes the form:

```
X-Weld-State: Pushed igniting_duck from weld frank

Changes were (in summary, topmost was applied last)

f0e6ceb Remove the earlier trailing comment
f00c9fc Add more trailing comments across the bases and to the weld
335718e One-duck: Also build one-duck, same as one
a75b292 One-duck: Add a comment to the end of the Makefile
```

- The header line is an `X-Weld-State: Pushed` line, in a different format from that used in the weld. It could be argued that it should
  1. not be an `X-Weld-State` line (although I don't think it can ever "escape" back into the weld and cause confusion)
  2. use a different term than `Pushed` (just in case it did "escape")
- This is the only place that the weld name (as taken from the XML file) is used (here it is `frank`) - is it actually useful, or should we be using something else?
- I quite like having the short-form SHA1 commit ids in there, since they do relate back to the weld repository, but it could be argued that they are not of use.

Do remember that it is always possible to do `weld push --edit` and edit this text before it is committed.

- Should we make `--edit` the default, and provide a `--no-edit` switch as well?

7.8 Output levels

Priority: low

We are probably still outputting too much text when `--verbose` is not specified.

We may be outputting too much or the wrong text when `--verbose` *is* specified.

I suspect we are not always outputting appropriate text (in order to be useful) when something goes wrong.

All of these need consideration.

7.9 (Over) use of git porcelain

Ideally we would use the git plumbing more, and git porcelain less, since the output of git porcelain is (in general) allowed to be a moving target.

7.10 Weld name

Priority: low

What is the weld name used for, if anything?

CHAPTER 8

The weld documentation: Sphinx and ReadTheDocs

8.1 Pre-built documentation

For your comfort and convenience, a pre-built version of the weld documentation is available at:

http://weld.readthedocs.org/

This is hosted by Read the Docs, who are wonderful people for providing such a facility. The documentation should get rebuilt on each push to the repository, which means that it should always be up-to-date.

8.2 Building the documentation

The weld documentation is built using Sphinx.

As said above, the easiest way to get the documentation is via Read the Docs, but if you want to build a copy yourself, then all you need to do is install Sphinx, and use the Makefile:

```
$ cd docs
$ make html
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search