Find the source for this document in the docs directory.

The ‘resources’ Project

2022-02-26 TWL imported from README markdown, still in process of conversion

What is it?

A set of scripts used for the projects found in 'github.com/thomas-walker-lynch' and other places where it has been adopted. These apply to C, C++, Python, Django, and Rust. This replaces any other script you might be accustom to for entering a ‘virtual environment’.

`setup`	sets up the environment for a project
`pull`	fancy shortcut for git pull
`push`	fancy shortcut for git push
`makefile`	a genetic makefile for C/C++ projects
`makefil-env`	default variables for make

These scripts have been evolving with each project, so tweaks might be required.

Assume we have a top level directory called 'repos', and we have expanded resources

```
/home/Thomas/repos/
  resources_repo/
    bin/<br>
    include/
    lib/
    media/
    tmp/
    LICENSE
    README.md
    projects-init.sh
```

Modifying the resources repo

If you make some changes and are testing them, then do that on a branch. Merge the branch after it is tested.

If you do something to the resources repo of general interest, then check it in to master.

If you need special modifications specific to your needs, i.e. not of general interest even to those who might come in the future and be working in the same language, etc., then make your very own branch.

If you have a fundamentally different way of going about this, some people want it, others don't - then that goes on a branch.

Your local directory for holding project, ‘repos’

1. make a top level directory for holding all the repos you work on, independent of language, etc.

   ```
   > mkdir ~/repos
   ```

2. expand the `system` repo, then inspect and install those script in the system

   ```
   > cd ~/repos
   > git clone git@github.com:Thomas-Walker-Lynch/system
   ```

   Follow the directions from the system repo for installing `home` and `Z`. There is not really much to it. ‘Z’ is used for timestamps. `home` returns the home directory from `/etc/passwd`.

3. expand the `resources` repo

   ```
   > cd ~/repos
   > git clone git@github.com:Thomas-Walker-Lynch/resources
   ```

4. add the following to `.bashrc`. Note the file `lib/bashrc` to get started.

   ```
   > export PATH=~/repos/resources/bin:"$PATH"
   ```

The setup script

This command will open a new shell with the environment in that shell setup for the project.

```
2022-02-25T13:06:24Z
lander10@beast §~/§
> setup Endian
Hello Emacs
/home/Thomas/repos/bin:/home/Thomas/repos/resources/bin:/usr/local/bin:/usr/bin:/bin

2022-02-25T13:06:28Z [Endian]
```

```
lander10@beast §~/repos/Endian§
>
```

Currently <tt>setup</tt> echos the path so that they user will understand where the executables come from.

The prompt carries sufficient information for making sense of the transcript when it is read later.

On the first line, the time shown is UTC in standard iso8601 format. This comes from the '/usr/bin/Z' script.

Following the time, in square brackets you will see the name of the project. If the square brackets are not there, then no project environment has been setup. If a user attempts to work on a project that has not been setup, he or she is likely to experience many errors.

On the second line we have the user name, machine name, and current working directory.

On the third line the familiar `>` appears. Anything you type after the prompt is taken as the command for the shell. It is conventional for script run as root user that this will instead be '#'.

The generic makefile

By default a C/C++ project wil have this directory structure:

```
~/repos/
    Endian/
        deprecated/
        doc/
        exec/
        lib/
        src/
        test/
        tmp/
        try/
        LICENSE
        makefile
        makefile-env
    resources/
    system/
    tmp/
```

Note <tt>makefile</tt> and <tt>makefile-env</tt> inside the Endian project are there only to override defaults on the <tt>resources</tt> versions of those files. They are stubs tweak a couple of things and then pass the heavy lifting to <tt>resources/lib/makefile</tt> and <tt>resources/lib/makefile-env</tt>. Leave the <tt>resources/lib/makefile</tt> and <tt>resources/lib/makefile-env</tt> where they are and do not make copies.

Give source files two suffixes. For example the Endian project source directory appears as:

```
2022-02-26T10:54:55Z [Endian]
thomas@beast § ~/resources §
> cd $PROJECT_HOME

2022-02-26T10:55:47Z [Endian]
thomas@beast § ~/Endian §
> ls src
endian.lib.c  endian_test.cli.c  ip4.lib.c  ip4_test.cli.c
```

Endian is a C project. For C++ sources the suffix will be <tt>.cc</tt>. I suppose the generic makefile should be improved by adding a variable to configure the string used for suffixes. Right now I think this convention is hard coded.

The compiled lib files will go into <tt>lib/$PROJECT_lib.a</tt> while the cli files will be made stand alone and appear in the <tt>exec</tt> directory. Presumably all the <tt>.cli.c</tt> files have a main function defined.

Push and pull scripts

These are git command sequence shortcuts. They do the usual things when pushing and pulling from a git repo and also do some checks.

Other things

Other useful things will be found in the resource tree.

Projects with submodules

By convention, projects with submodules have the suffix <tt>_ensemble</tt>.

1. install the project

Suppose the project with submodules is called <project>_ensemble

```
> cd ~/repos
> git clone git@github.com/<user>/<project>_ensemble.git
> mv <project>_ensemble <project>_master
> cd <project>_master
```

Note in the line that moves the cloned repo directory to `<project>_master`, you might use a different suffix than `master`. Conventionally the suffix is the branch to be checked out and worked on, but the scripts do not care what it is set to. Inside the scripts this suffix is called the project 'version'. If this checkout is for working on multiple branches, just leave the <tt>_ensemble</tt>.

2. for submodules that have not yet been added:

```
> git submodule add https://github.com/../<resource_project>
> git submodule add https://github.com/../<resource_project>
...
```

Etc. for the other modules

3. if a submodule is empty, then do the following:

```
> git submodule init
> git submodule update
```

Submodules directories will be empty when the `--recursive` switch is not provided with the clone. Actually, I prefer not to use `--recursive` and then to follow up with an `init` and `update` so that it is easier to tell what caused errors.

generally about project security

0. Contain your development work

Consdider giving the repos directory its own user. If you do add this link so that the scripts can still have a 'repos' target.

```
> ln -s . repos
```

Alternatively put them in a container or a virtual machine.

1. audit source files and legible scripts

Watch what git pulls. For new things that are source code or scripts, audit those.

Note! '.gitignore' does not apply to pulled content.

2. hidden files

Files that should be audited must not be hidden. Otherwise they are too easy to miss during the audit.

Hidden files are not seen by file globs, so does not see the <tt>.gitignore</tt> file.

```
> ls -ld *
drwxrwx---. 1 thomas thomas    64 2022-02-26 07:55 bin
drwxrwx---. 1 thomas thomas   142 2022-02-26 07:03 lib
-rw-rw----. 1 thomas thomas  1076 2022-02-25 07:25 LICENSE
drwxrwx---. 1 thomas thomas    20 2022-02-25 07:25 media
-rw-rw----. 1 thomas thomas   141 2022-02-25 07:25 projects-init.sh
-rw-rw----. 1 thomas thomas 18209 2022-02-26 08:02 README.md
drwxrwx---. 1 thomas thomas    84 2022-02-26 05:40 tmp
```

Globs not seeing hidden files is generally considered to be a good thing when we do not want usual shell operations
to see it. For example <tt> cp * </tt> a repo manually we often want a copy of the project, not a copy of the repo
administrative files. There is a separate command for copying a repo, <tt> git clone </tt>.

If we want to 'unhide' dot file from a file glob, then we rename it. That is best. In the case <tt>.gitignore</tt>
<tt>git</tt> is expecting to see the file with that very name. If we make a file that is not hidden and then
link <tt>.gitignore</tt> to it, git will yell at us "too many symbolic links!".

Hence, we can expose a file like <tt>.gitignore</tt> by linking to it (rather than the other way around).

When doing this, typically the link name is the same but without the dot in front of it. Here I expose the
<tt>.gitignore</tt>.

```
2022-02-26T13:02:28Z [Endian]
thomas@beast § ~/resources §
> ln -s .gitignore gitignore
```

And now the file glob will pick it up.

```
2022-02-26T13:07:13Z [Endian]
thomas@beast § ~/resources §
> ls -ld *
drwxrwx---. 1 thomas thomas    64 2022-02-26 07:55 bin
lrwxrwxrwx. 1 thomas thomas    10 2022-02-26 08:07 gitignore -> .gitignore
drwxrwx---. 1 thomas thomas   142 2022-02-26 07:03 lib
-rw-rw----. 1 thomas thomas  1076 2022-02-25 07:25 LICENSE
drwxrwx---. 1 thomas thomas    20 2022-02-25 07:25 media
-rw-rw----. 1 thomas thomas   141 2022-02-25 07:25 projects-init.sh
-rw-rw----. 1 thomas thomas 20454 2022-02-26 08:14 README.md
drwxrwx---. 1 thomas thomas    84 2022-02-26 05:40 tmp
```

2. PATH

Do not put a repo in your execution PATH. When running tests and such call them out locally, <tt>./program</tt>,
or through one of the environment variables initialized by the <tt>setup</tt>.

The <tt>~/repos/resources/bin</tt> directory is an exception to this. We need to maintain the executables used for
maintaing the projects themselves. Be careful to audit any changes to this <tt>resources</tt>.

3. pulled binary executbles

It is best to not pull binary executables then run them, because they can not be audited.

It sometimes happens that a co-developers will compile and create an executable, and then not clean it, and then
accidentally the user poisons the repo with them. Watch your pulls and clones, and remove executables from the

repo, and complain about them. The generic makefile, push, and pull scripts can help with this.

When an unauditable executable arrives in a pull it creates a trust problem. We distrist both the source and the integrity of the distribution process. This latter problem can be mitigated if cryptographic signatures are independently available.

Consider running such an executable in a container.

Make sure make your clean targets remove executables before a push.

Be careful with the programs in <tt>resources/bin</tt>.

4. check for the .gitignore and audit it, add to it

If you add the <tt>exec</tt> or <tt>bin</tt> directories to gitignore and then forget to run make clean, it might prevent accidentally adding executables.

Typical .gitignore files:

```
env/
tmp/
.*
!.gitignore
*~
```

For a C or C++ project home directory we will also ignore various intermediate files,

```
tmp/
bin/
exec/
.*
!.gitignore
*~
*.o
*.i
*.s
a.out
```

For a python project:

```
tmp/
```

```
.*
!.gitignore
__pycache__/
**/*.pyc
```

And for a django project:

```
tmp/
.*
!.gitignore
__pycache__/
**/*.pyc
manage.py
**/migrations
.vscode
```

General info and concepts

What is a ‘project’

The term ‘project’ occurs repeatedly in this document and generally when we talk about code, so it is best to start by nailing that term down. Generally speaking, a ‘project’ has a well defined final product, along with a list resource needs, tasks, task dependencies, and a tasks schedule. It is the job of the team members to execute said tasks.

In project management parlance, a group of related projects is called a ‘program’. Well that sure is an unfortunate choice of terminology for us CS people, so instead we will call a group of related projects a ‘project ensemble’. A project ensemble itself is also a kind of project, where the tasks are the component projects. Hence we have a recursive structure. CS people like recursive structures ;-)

Projects are held in git repositories, as project ensembles are also projects, they are also held in git repositories. When we clone a project ensemble we will find other git repositories have been expanded inside the

directory tree. Those are for the component projects. In git speak we call these component project directory trees
'submodules'.

When a project ensemble is expanded out, we end up with a directory tree structure where project resources, tools, and
work products are stored.

Where project code goes

On a typical project we will have three distinct types of code:

1. the application source code
2. the libraries and other resources the application makes use of
3. the tools used for building the source code

Let's give these code types short names:

1. source
2. resources
3. tools

We have various places where we might put code that we need in a program:

1. in the system
2. in the developer's user directory
3. in a project ensemble
4. in a project's home directory.

We shorten this list of places to:

1. system
2. user
3. ensemble
4. project

Now combining our code and locations into one list:

1. source
   1. project
2. resources
   1. system
   2. user
   3. ensemble
3. tools
   1. system
   2. user
   3. ensemble

So there is only one place we will find the application source code that we are developing, and that is under the project directory. Resources that we may need in order to compile our code will be found either in the system, the user directory, or in the ensemble. The same can be said for the tools we will use.

This is what my home directory looks like:

```
/home/thomas/
  bin/
  Desktop/
  Documents/
  Downloads/
  repos/
    chessgame/
    resources/  <--- resources for all projects - expand this repo here
        LICENSE
        README.md
        bin/
            makefile-cc
            pull
            push
            rm_tilda_files_tree
            setup
    subu/
    tm/
    ws4_master/   <--- an ensemble directory
      LICENSE
      README.md
      env/  <--- resources specific to the ensemble (wish this was called ‘resources’
          bin/
          lib/
          include/
```

```
tmp/
uWebSockets/ <--- resource project directory
ws4/   <--- target project directory
```

Now looking under my `projects` directory, and expanding out `ws4_master`:

```
ws4_master/
  LICENSE
  README.md
  env/
     bin/
     include/
     lib/
  tmp/
  uWebSockets/
  ws4/
```

`ws4_master` is a project ensemble. The component projects include `uWebSockets` and `ws4`. This project ensemble also comes from a git repo.

`ws4_master/ws4` is the home directory for the project this team is actively working on. We know it is the active project because it has the same name as the ensemble prefix. Whereas `ws4_master/uWebSockets` is someone else's github project. There are other people working that, but it is not us, and it is being developed in a different environment. Rather we are just making use of the work product of that project.

Also inside of `ws4_master` we have a directory called `env`. Frankly, I don't like the name. I have toyed with calling it `project-share`, but `env` is the name that Python and others expect. Perhaps make it a symbol link? This directory is used to hold project specific resources and tools. Note that the contents of `env` are *not* pushed to the repo. This means that custom edits you make to scripts will not be backed up to the repo. I also do not like `env` because it is not pushed to the repo, but it might be *pulled* from it. `.gitignore` does not affect pulls. This is a security hazard.

Repo and Directory Naming (again)

If a repo has submodules in it, I generally give the repo name a suffix of '_ensemble'.

After cloning, if I am only going to work on a given breanch within that clone, I will change the suffix to the branch name, <project>_<branch>.

So you ask why do we need more than one directory for the same repo? Well in this case I am running a web server against the v2.0 branch, and it needs to see the files the v2.0. The v1.0 directory was the one that was formally being served. There are no servers pointed at it now, so I should probably delete it. If I ever need v1.0 again I can always check it out.

```
> cd ~/repos
> ls
    customer_gateway_master
    customer_gateway_v1.0
    customer_gateway_v2.0
```

This is how the `customer_gateway_master` project environment directory was made:

```
> cd ~/repos
> git clone --recursive --jobs 8
git@github.com:Reasoning-Technology/customer_gateway_ensemble.git
> mv customer_gateway_ensemble customer_gateway_master
```

When downloading a `<project>_ensemble` repo, we can expand the submodules at the same time
by including the --recursive switch, as was shown above. If the `--recursive` switch is not given, the submodules will have to be initialized and updated.

The second directory was then created with the commands:

```
> git clone --recursive --jobs 8
git@github.com:Reasoning-Technology/customer_gateway_ensemble.git
> mv customer_gateway_ensemble customer_gateway_v1.0
> cd customer_gateway_v1.0
> git checkout v1.0
```

Cloning a `git` repository produces a directory tree, which in git speak is apparently called a `module`.

We may `cd` into a module and clone another module, this will be called a ‘submodule’. A submodule clone operation requires a special command so that the module will know it is there:

```
git submodule add <repo>
```

We might do this because our project depends on other projects, and those other projects have their own git repositories, or because we are developing more than one project together with one shared environment.

We then use the submodule just like we would use any other git module. I.e. after we make changes we must add the changes, then commit them, and then push the submodule.

The parent module only sees the submodule changes when there is a commit in the submodule. Hence, after there is a commit in a submodule, we must go up to the module, and then add the submodule, commit, then push the module.

We truly have two layers, and we have to maintain them individually. Luckily we have some scripts so that we don't to type stuff twice. In the `project-share` project there are two scripts, one called `push` the other called `pull`. When we run the `push` script it goes into the project home and does an add, commit, and push. It then goes up to the ensemble directory and `git add`s the project submodule, commits the change, and then pushes. Finally it pops directory back to the project. The current `pull` script pulls down all the submodules and the ensemble.