Manage multiple Git repositories with this script

GTWS is a set of scripts that make it easy to have development environments for different projects and different versions of a project.

Great Teeming Workspaces (GTWS) is a complex workspace management package for Git that makes it easy to have development environments for different projects and different versions of a project.

More on Git

- What is Git?
- Git cheat sheet
- Markdown cheat sheet
- New Git articles

Somewhat like Python venv, but for languages other than Python, GTWS handles workspaces for multiple versions of multiple projects. You can create, update, enter, and leave workspaces easily, and each project or version combination has (at most) one local origin that syncs to and from the upstream—all other workspaces update from the local origin.

Layout

```
${GTWS_ORIGIN}/<project>/<repo>[/<version>]
${GTWS_BASE_SRCDIR}/<project>/<version>/<workspacename>/{<repo>[,<repo>...]}
```

Each level in the source tree (plus the homedir for globals) can contain a **.gtwsrc** file that maintains settings and Bash code relevant to that level. Each more specific level overrides the higher levels.

Setup

Check out GTWS with:

```
git clone https://github.com/dang/gtws.git
```

Set up your **${HOME}/.gtwsrc**. It should include **GTWS_ORIGIN** and optionally **GTWS_SETPROMPT**.

Add the repo directory to your path:

```
export PATH="${PATH}:/path/to/gtws
```

Configuration

Configuration is via cascading **.gtwsrc** files. It walks the real path down from the root, and each **.gtwsrc** file it finds is sourced in turn. More specific files override less specific files.

Set the following in your top-level **~/.gtws/.gtwsrc**:

- **GTWS_BASE_SRCDIR:** This is the base of all the projects' source trees. It defaults to **$HOME/src**.
- **GTWS_ORIGIN:** This sets the location of the origin Git trees. It defaults to **$HOME/origin**.
- **GTWS_SETPROMPT:** This is optional. If it's set, the shell prompt will have the workspace name in it.
- **GTWS_DEFAULT_PROJECT:** This is the project used when no project is given or known. If it is not given, projects must be specified on the command line.
- **GTWS_DEFAULT_PROJECT_VERSION:** This is the default version to check out. It defaults to **master**.

Set the following at the project level of each project:

- **GTWS_PROJECT:** The name (and base directory) of the project.
- **gtws_project_clone:** This function is used to clone a specific version of a project. If it is not defined, then it is assumed that the origin for the project contains a single directory per version, and that contains a set of Git repos to clone.
- **gtws_project_setup:** This optional function is called after all cloning is done and allows any additional setup necessary for the project, such as setting up workspaces in an IDE.

Set this at the project version level:

- **GTWS_PROJECT_VERSION:** This is the version of the project. It's used to pull from the origin correctly. In Git, this is likely the branch name.

These things can go anywhere in the tree and can be overridden multiple times, if it makes sense:

- **GTWS_PATH_EXTRA:** These are extra path elements to be added to the path inside the workspace.
- **GTWS_FILES_EXTRA:** These are extra files not under version control that should be copied into each checkout in the workspace. This includes things like **.git/info/exclude**, and each file is relative to the base of its repo.

Origin directories

GTWS_ORIGIN (in most scripts) points to the pristine Git checkouts to pull from and push to.

Layout of **${GTWS_ORIGIN}**:

- **/<project>**
  - This is the base for repos for a project.
  - If **gtws_project_clone** is given, this can have any layout you desire.
  - If **gtws_project_clone** is not given, this must contain a single subdirectory named **git** that contains a set of bare Git repos to clone.

Workflow example

Suppose you have a project named **Foo** that has an upstream repository at **github.com/foo/foo.git**. This repo has a submodule named **bar** with an upstream

at **github.com/bar/bar.git**. The Foo project does development in the master branch and uses stable version branches.

Before you can use GTWS with Foo, first you must set up the directory structure. These examples assume you are using the default directory structure.

- Set up your top level **.gtwsrc**:
  - **cp ${GTWS_LOC}/examples/gtwsrc.top ~/.gtwsrc**
  - Edit **~/.gtwsrc** and change as necessary.
- Create top-level directories:
  - **mkdir -p ~/origin ~/src**
- Create and set up the project directory:
  - **mkdir -p ~/src/foo**
    cp ${GTWS_LOC}/examples/gtwsrc.project ~/src/foo/.gtwsrc
  - Edit **~/src/foo/.gtwsrc** and change as necessary.
- Create and set up the master version directory:
  - **mkdir -p ~/src/foo/master**
    cp ${GTWS_LOC}/examples/gtwsrc.version ~/src/foo/master/.gtwsrc
  - Edit **~/src/foo/master/.gtwsrc** and change as necessary.
- Go to the version directory and create a temporary workspace to set up the mirrors:
  - **mkdir -p ~/src/foo/master/tmp**
    cd ~/src/foo/master/tmp
    git clone --recurse-submodules git://github.com/foo/foo.git
    cd foo
    gtws-mirror -o ~/origin -p foo
  - This will create **~/origin/foo/git/foo.git** and **~/origin/foo/submodule/bar.git**.
  - Future clones will clone from these origins rather than from upstream.
  - This workspace can be deleted now.

At this point, work can be done on the master branch of Foo. Suppose you want to fix a bug named **bug1234**. You can create a workspace for this work to keep it isolated from anything else you're working on, and then work within this workspace.

- Go to the version directory, and create a new workspace:
  - **cd ~/src/foo/master**
    mkws bug1234
  - This creates **bug1234/**, and inside it checks out Foo (and its submodule **bar**) and makes **build/foo** for building it.
- Enter the workspace. There are two ways to do this:
  - **cd ~/src/foo/master/bug1234**
    startws
    or
    cd ~/src/foo/master/
    startws bug1234
  - This starts a subshell within the bug1234 workspace. This shell has the GTWS environment plus any environment you set up in your stacked **.gtwsrc** files. It also adds the base of the workspace to your CD path, so you can **cd** into relative paths from that base.
  - At this point, you can do work on bug1234, build it, test it, and commit your changes. When you're ready to push to upstream, do this:

cd foo
wspush

- **wspush** will push the branch associated with your workspace—first to your local origin and then to the upstream.
- If upstream changes. you can sync your local checkout using:
  git sync
- This envokes the **git-sync** script in GTWS, which will update your checkout from the local origin. To update the local origin, use:
  git sync -o
- This will update your local origin and submodules' mirrors, then use those to update your checkout. **git-sync** has other nice features.
- When you're done using the workspace, just exit the shell:
  exit
- You can re-enter the workspace at any time and have multiple shells in the same workspace at the same time.

- When you're done with a workspace, you can remove it using the **rmws** command or just remove its directory tree.
- There is a script named **tmws** that enters a workspace within tmux, creating a set of windows/panes that are fairly specific to my workflow. Feel free to modify it to suit your needs.

The script

```
#!/bin/bash
# Functions for gtws
#

GTWS_LOC=$(readlink -f $(dirname "${BASH_SOURCE[0]}"))
export GTWS_LOC

# if is_interactive; then echo "interactive" fi
#
# Check for an interactive shell
is_interactive() {
        case $- in
                *i*)
                        # Don't die in interactive shells
                        return 0
                        ;;
                *)
                        return 1
                        ;;
        esac
}

# if can_die; then exit
#
# Check to see if it's legal to exit during die
can_die() {
        if (( BASH_SUBSHELL > 0 )); then
                debug_print "\t\tbaby shell; exiting"
                return 0
        fi
        if ! is_interactive; then
                debug_print "\t\tNot interactive; exiting"
                return 0
        fi
        debug_print "\t\tParent interactive; not exiting"
        return 1
}
```

```
# In a function:
# command || die "message" || return 1
# Outside a function:
# command || die "message"
#
# Print a message and exit with failure
die() {
        echo -e "Failed: $1" >&2
        if [ ! -z "$(declare -F | grep "GTWScleanup")" ]; then
                GTWScleanup
        fi
        if can_die; then
                exit 1
        fi
        return 1
}

# Alternativess for using die properly to handle both interactive and script useage:
#
# Version 1:
#
#testfunc() {
#       command1 || die "${FUNCNAME}: command1 failed" || return 1
#       command2 || die "${FUNCNAME}: command2 failed" || return 1
#       command3 || die "${FUNCNAME}: command3 failed" || return 1
#}
#
# Version 2:
#
#testfunc() {
#       (
#               command1 || die "${FUNCNAME}: command1 failed"
#               command2 || die "${FUNCNAME}: command2 failed"
#               command3 || die "${FUNCNAME}: command3 failed"
#       )
#       return $?
#}
#
# Optionally, the return can be replaced with this:
#       local val=$?
#       [[ "${val}" == "0" ]] || die
#       return ${val}
# This will cause the contaning script to abort

# usage "You need to provide a frobnicator"
#
# Print a message and the usage for the current script and exit with failure.
usage() {
        local myusage;
        if [ -n "${USAGE}" ]; then
                myusage=${USAGE}
        else
                myusage="No usage given"
        fi
        local me;
        if [ -n "${ME}" ]; then
                me=${ME}
        else
                me=$(basename $0)
        fi
        if [ -n "$1" ]; then
                echo "$@"
        fi
        echo ""
        if [ -n "${DESCRIPTION}" ]; then
                echo -e "${me}: ${DESCRIPTION}"
                echo ""
        fi
        echo "Usage:"
        echo "${me} ${myusage}"
```

```
        if [ -n "${LONGUSAGE}" ]; then
                echo -e "${LONGUSAGE}"
        fi
        exit 1
}

# debug_print "Print debug information"
#
# Print debug information based on GTWS_VERBOSE
debug_print() {
        if [ -n "${GTWS_VERBOSE}" ]; then
                echo -e "${GTWS_INDENT}$@" >&2
        fi
}

# debug_trace_start
#
# Start tracing all commands
debug_trace_start() {
        if [ -n "${GTWS_VERBOSE}" ]; then
                set -x
        fi
}

# debug_trace_stop
#
# Stop tracing all commands
debug_trace_stop() {
        set +x
}

# cmd_exists ${cmd}
#
# Determine if a command exists on the system
function cmd_exists {
        which $1 > /dev/null 2>&1
        if [ "$?" == "1" ]; then
                die "You don't have $1 installed, sorry" || return 1
        fi
}

# is_git_repo ${dir}
#
# return success if ${dir} is in a git repo, or failure otherwise
is_git_repo() {
        debug_print "is_git_repo $1"
        if [[ $1 == *:* ]]; then
                debug_print "    remote; assume good"
                return 0
        elif [ ! -d "$1" ]; then
                debug_print "    fail: not dir"
                return 1
        fi
        cd "$1"
        git rev-parse --git-dir >/dev/null 2>&1
        local ret=$?
        cd - > /dev/null
        debug_print "    retval: $ret"
        return $ret
}

# find_git_repo ${basedir} ${repo_name} repo_dir
#
# Find the git repo for ${repo_name} in ${basedir}.  It's one of ${repo_name}
# or ${repo_name}.git
#
# Result will be in the local variable repo_dir  Or:
#
# repo_dir=$(find_git_repo ${basedir} ${repo_name})
#
```

```
function find_git_repo {
        local basedir=$1
        local repo_name=$2
        local __resultvar=$3
        local try="${basedir}/${repo_name}"

        if ! is_git_repo "${try}" ; then
                try=${try}.git
        fi

        is_git_repo "${try}" || die "${repo_name} in ${basedir} is not a git repository" || return 1

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$try'"
        else
                echo "$try"
        fi
}

# git_top_dir top
#
# Get the top level of the git repo contaning PWD, or return failure;
#
# Result will be in local variable top  Or:
#
# top = $(git_top_dir)
#
# Result will be in local variable top
function git_top_dir {
        local __resultvar=$1
        local __top="$(git rev-parse --show-toplevel 2>/dev/null)"

        if [ -z "${__top}" ]; then
                die "${PWD} is not a git repo" || return 1
        fi
        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__top'"
        else
                echo "$__top"
        fi
}

# is_git_rebase
#
# return success if git repo is in a rebase
is_git_rebase() {
        debug_print "is_git_rebase $1"
        (test -d "$(git rev-parse --git-path rebase-merge)" || \
                test -d "$(git rev-parse --git-path rebase-apply)" )
        local ret=$?
        debug_print "    retval: $ret"
        return $ret
}

# is_docker
#
# return success if process is running inside docker
is_docker() {
        debug_print "is_docker"
        grep -q docker /proc/self/cgroup
        return $?
}

# is_gtws
#
# return success if process is running inside a workspace
is_gtws() {
        if [ -n "${GTWS_WS_GUARD}" ]; then
                return 0
        fi
```

```
	return 1
}

function gtws_rcp {
	rsync --rsh=ssh -avzS --progress --ignore-missing-args --quiet "$@"
}

function gtws_cpdot {
	local srcdir=$1
	local dstdir=$2

	debug_print "${FUNCNAME} - ${srcdir} to ${dstdir}"
	if [ -d "${srcdir}" ] && [ -d "${dstdir}" ]; then
		shopt -s dotglob
		cp -a "${srcdir}"/* "${dstdir}"/
		shopt -u dotglob
	fi
}

# gtws_find_dockerfile dockerfile
#
# Result will be in local variable dockerfile  Or:
#
# dockerfile = $(gtws_find_dockerfile)
#
# Result will be in local variable dockerfile
#
# Get the path to the most-specific Dockerfile
function gtws_find_dockerfile {
	local __resultvar=$1
	local __dir="${GTWS_WSPATH}"
	local __file="Dockerfile"

	debug_print "${FUNCNAME} - trying ${__dir}/${__file}"
	if [ ! -f "${__dir}/${__file}" ]; then
		# Version dir
		__dir=$(dirname "${__dir}")
		debug_print "${FUNCNAME} - trying ${__dir}/${__file}"
	fi
	if [ ! -f "${__dir}/${__file}" ]; then
		# Project dir
		__dir=$(dirname "${__dir}")
		debug_print "${FUNCNAME} - trying ${__dir}/${__file}"
	fi
	if [ ! -f "${__dir}/${__file}" ]; then
		# Top level, flavor
		__dir="${GTWS_LOC}/dockerfiles"
		__file="Dockerfile-${FLAVOR}"
		debug_print "${FUNCNAME} - trying ${__dir}/${__file}"
	fi
	if [ ! -f "${__dir}/${__file}" ]; then
		# Top level, base
		__dir="${GTWS_LOC}/dockerfiles"
		__file="Dockerfile-base"
		debug_print "${FUNCNAME} - trying ${__dir}/${__file}"
	fi
	if [ ! -f "${__dir}/${__file}" ]; then
		die "Could not find a Dockerfile" || return 1
	fi

	debug_print "${FUNCNAME} - found ${__dir}/${__file}"
	if [[ "$__resultvar" ]]; then
		eval $__resultvar="'${__dir}/${__file}'"
	else
		echo "$__dir"
	fi
}

# gtws_smopvn ${GTWS_SUBMODULE_ORIGIN:-${GTWS_ORIGIN}} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION} ${GTWS_WSNAME} smopvn
#
```

```
# Result will be in local variable smopvn.  Or:
#
# smopvn = $(gtws_smopvn ${GTWS_SUBMODULE_ORIGIN:-${GTWS_ORIGIN}} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION} ${GTWS_WSNAME}
#
# Result will be in local variable smovpn
#
# Get the path to submodules for this workspace
function gtws_smopvn {
        local origin=$1
        local project=$2
        local version=$3
        local name=$4
        local __resultvar=$5
        local __smopv="${origin}/${project}/submodule"

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__smopv'"
        else
                echo "$__smopv"
        fi
}

# gtws_opvn ${GTWS_ORIGIN} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION} ${GTWS_WSNAME} opvn
#
# Result will be in local variable opvn.  Or:
#
# opvn = $(gtws_opvn ${GTWS_ORIGIN} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION} ${GTWS_WSNAME})
#
# Result will be in local variable opvn.
#
# Get the path to git repos for this workspace
function gtws_opvn {
        local origin=$1
        local project=$2
        local version=$3
        local name=$4
        local __resultvar=$5
        local __opv="${origin}/${project}/${version}"

        if [[ $__opv == *:* ]]; then
                __opv="${__opv}/${name}"
                debug_print "remote; using opvn $__opv"
        elif [ ! -d "${__opv}" ]; then
                __opv="${origin}/${project}/git"
                if [ ! -d "${__opv}" ]; then
                        die "No opvn for ${origin} ${project} ${version}" || return 1
                fi
        fi
        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__opv'"
        else
                echo "$__opv"
        fi
}

# gtws_submodule_url ${submodule} url
#
# Result will be in local variable url  Or:
#
# url = $(gtws_submodule_url ${submodule})
#
# Result will be in local variable url
#
# Get the URL for a submodule
function gtws_submodule_url {
        local sub=$1
        local __resultvar=$2
        local __url=$(git config --list | grep "submodule.*url" | grep "\<${sub}\>" | cut -d = -f 2)

        if [ -z "${__url}" ]; then
```

```
                local rpath=${PWD}
                local subsub=$(basename "${sub}")
                cd "$(dirname "${sub}")"
                debug_print "${FUNCNAME} trying ${PWD}"
                __url=$(git config --list | grep submodule | grep "\<${subsub}\>" | cut -d = -f 2)
                cd "${rpath}"
        fi

        debug_print "${FUNCNAME} $sub url: $__url"
        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__url'"
        else
                echo "$__url"
        fi
}

# gtws_submodule_mirror ${smopv} ${submodule} ${sub_sub_basename} mloc
#
# Result will be in local variable mloc  Or:
#
# mloc = $(gtws_submodule_mirror ${smopv} ${submodule} ${sub_sub_basename})
#
# Result will be in local variable mloc
#
# Get the path to a local mirror of the submodule, if it exists
function gtws_submodule_mirror {
        local smopv=$1
        local sub=$2
        local sub_sub=$3
        local __resultvar=$4
        local __mloc=""
        local url=$(gtws_submodule_url ${sub})
        if [ -n "${url}" ]; then
                local urlbase=$(basename ${url})
                # XXX TODO - handle remote repositories
                #if [[ ${smopv} == *:* ]]; then
                ## Remote SMOPV means clone from that checkout; I don't cm
                #refopt="--reference ${smopv}/${name}/${sub}"
                if [ -d "${smopv}/${urlbase}" ]; then
                        __mloc="${smopv}/${urlbase}"
                fi
        fi

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__mloc'"
        else
                echo "$__mloc"
        fi
}

# gtws_submodule_paths subpaths
#
# Result will be in local variable subpaths  Or:
#
# subpaths = $(gtws_submodule_paths)
#
# Result will be in local variable subpaths
#
# Get the paths to submodules in a get repo.  Does not recurse
function gtws_submodule_paths {
        local __resultvar=$1
        local __subpaths=$(git submodule status | sed 's/^ *//' | cut -d ' ' -f 2)

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__subpaths'"
        else
                echo "$__subpaths"
        fi
}
```

```
# gtws_submodule_clone [<base-submodule-path>] [<sub-sub-basename>]
#
# This will set up all the submodules in a repo.  Should be called from inside
# the parent repo
function gtws_submodule_clone {
	local smopv=$1
	local sub_sub=$2
	local sub_paths=$(gtws_submodule_paths)
	local rpath="${PWD}"

	if [ -z "${smopv}" ]; then
		smopv=$(gtws_smopvn "${GTWS_SUBMODULE_ORIGIN:-${GTWS_ORIGIN}}" "${GTWS_PROJECT}" "${GTWS_PROJECT_VERSION
	fi
	git submodule init || die "${FUNCNAME}: Failed to init submodules" || return 1
	for sub in ${sub_paths}; do
		local refopt=""
		local mirror=$(gtws_submodule_mirror "${smopv}" "${sub}" "${sub_sub}")
		debug_print "${FUNCNAME} mirror: ${mirror}"
		if [ -n "${mirror}" ]; then
			refopt="--reference ${mirror}"
		fi
		git submodule update ${refopt} "${sub}"
		# Now see if there are recursive submodules
		cd "${sub}"
		gtws_submodule_clone "${smopv}/${sub}_submodule" "${sub}" || return 1
		cd "${rpath}"
	done
}

# gtws_repo_clone <base-repo-path> <repo> <branch> [<base-submodule-path>] [<target-directory>]
function gtws_repo_clone {
	local baserpath=${1%/}
	local repo=$2
	local branch=$3
	local basesmpath=$4
	local rname=${5:-${repo%.git}}
	local rpath="${baserpath}/${repo}"
	local origpath=${PWD}

	if [[ ${rpath} != *:* ]]; then
		if [ ! -d "${rpath}" ]; then
			rpath="${rpath}.git"
		fi
	fi
	if [ -z "${basesmpath}" ]; then
		basesmpath="${baserpath}"
	fi
	debug_print "${FUNCNAME}: cloning ${baserpath} - ${repo} : ${branch} into ${GTWS_WSNAME}/${rname} submodules: ${

	# Main repo
	#git clone --recurse-submodules -b "${branch}" "${rpath}" || die "failed to clone ${rpath}:${branch}" || return
	git clone -b "${branch}" "${rpath}" ${rname} || die "${FUNCNAME}: failed to clone ${rpath}:${branch}" || return

	# Update submodules
	cd "${rname}" || die "${FUNCNAME}: failed to cd to ${rpath}" || return 1
	gtws_submodule_clone "${basesmpath}" || return 1
	cd "${origpath}" || die "${FUNCNAME}: Failed to cd to ${origpath}" || return 1

	# Copy per-repo settings, if they exist
	gtws_cpdot "${baserpath%/git}/extra/repo/${rname}" "${origpath}/${rname}"

	# Extra files
	for i in ${GTWS_FILES_EXTRA}; do
		local esrc=

		IFS=':' read -ra ARR <<< "$i"
		if [ -n "${ARR[1]}" ]; then
			dst="${rname}/${ARR[1]}"
		else
			dst="${rname}/${ARR[0]}"
```

```
                fi

                if [ -n "${GTWS_REMOTE_IS_WS}" ]; then
                        esrc="${baserpath}/${dst}"
                else
                        esrc="${baserpath%/git}"
                fi

                gtws_rcp "${esrc}/${ARR[0]}" "${dst}"
        done
}

# gtws_project_clone_default ${GTWS_ORIGIN} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION} ${GTWS_WSNAME} [${SUBMODULE_BASE}]
#
# Clone a version of a project into ${GTWS_WSPATH} (which is the current working directory).  This is the default versio
function gtws_project_clone_default {
        local origin=$1
        local project=$2
        local version=$3
        local name=$4
        local basesmpath=$5
        local opv=$(gtws_opvn "${origin}" "${project}" "${version}" "${name}")
        local wspath=${PWD}
        local repos=
        local -A branches

        if [ -z "${GTWS_PROJECT_REPOS}" ]; then
                for i in "${opv}"/*; do
                        repos="$(basename $i) $repos"
                        branches[$i]=${version}
                done
        else
                for i in ${GTWS_PROJECT_REPOS}; do
                        IFS=':' read -ra ARR <<< "$i"
                        repos="${ARR[0]} $repos"
                        if [ -n "${ARR[1]}" ]; then
                                branches[${ARR[0]}]=${ARR[1]}
                        else
                                branches[${ARR[0]}]=${version}
                        fi
                done
        fi

        if [ -z "${basesmpath}" ] || [ ! -d "${basesmpath}" ]; then
                basesmpath="${opv}"
        fi

        for repo in ${repos}; do
                gtws_repo_clone "${opv}" "${repo}" "${branches[${repo}]}" "${basesmpath}"
        done

        # Copy per-WS settings, if they exist
        gtws_cpdot "${opv%/git}/extra/ws" "${wspath}"
}

# gtws_repo_setup ${wspath} ${repo_path}
#
# The project can define gtws_repo_setup_local taking the same args to do
# project-specific setup.  It will be called last.
#
# Post-clone setup for an individual repo
function gtws_repo_setup {
        local wspath=$1
        local rpath=$2
        local savedir="${PWD}"

        if [ ! -d "${rpath}" ]; then
                return 0
        fi
```

```
        cd "${rpath}/src" 2>/dev/null \
                || cd ${rpath} \
                || die "Couldn't cd to ${rpath}" || return 1

        maketags ${GTWS_MAKETAGS_OPTS} > /dev/null 2> /dev/null &

        cd ${wspath} || die "Couldn't cd to ${wspath}" || return 1

        mkdir -p "${wspath}/build/$(basename ${rpath})"

        cd "${savedir}"

        if [ -n "$(declare -F | grep "\<gtws_repo_setup_local\>")" ]; then
                gtws_repo_setup_local "${wspath}" "${rpath}" \
                        || die "local repo setup failed" || return 1
        fi
}

# gtws_project_setup${GTWS_WSNAME} ${GTWS_ORIGIN} ${GTWS_PROJECT} ${GTWS_PROJECT_VERSION}
#
# The project can define gtws_project_setup_local taking the same args to do
# project-specific setup.  It will be called last.
#
# Post clone setup of a workspace in ${GTWS_WSPATH} (which is PWD)
function gtws_project_setup {
        local wsname=$1
        local origin=$2
        local project=$3
        local version=$4
        local wspath=${PWD}
        local opv=$(gtws_opvn "${origin}" "${project}" "${version}" "placeholder")

        for i in "${wspath}"/*; do
                gtws_repo_setup "${wspath}" "${i}"
        done

        mkdir "${wspath}"/install
        mkdir "${wspath}"/chroots
        mkdir "${wspath}"/patches

        if [ -n "$(declare -F | grep "\<gtws_project_setup_local\>")" ]; then
                gtws_project_setup_local "${wsname}" "${origin}" "${project}" \
                        "${version}" || die "local project setup failed" || return 1
        fi
}

# load_rc /path/to/workspace
#
# This should be in the workspace-level gtwsrc file
# Recursively load all RC files, starting at /
function load_rc {
        local BASE=$(readlink -f "${1}")
        # Load base RC first
        debug_print "load_rc: Enter + Top: ${BASE}"
        source "${HOME}"/.gtwsrc
        while [ "${BASE}" !=  "/" ]; do
                if [ -f "${BASE}"/.gtwsrc ]; then
                        load_rc "$(dirname ${BASE})"
                        debug_print "\tLoading ${BASE}/.gtwsrc"
                        source "${BASE}"/.gtwsrc
                        return 0
                fi
                BASE=$(readlink -f $(dirname "${BASE}"))
        done
        # Stop at /

        return 1
}

# clear_env
```

```
#
# Clear the environment of GTWS_* except for the contents of GTWS_SAVEVARS.
# The default values for GTWS_SAVEVARS are below.
function clear_env {
        local savevars=${GTWS_SAVEVARS:-"LOC PROJECT PROJECT_VERSION VERBOSE WSNAME"}
        local verbose="${GTWS_VERBOSE}"
        debug_print "savevars=$savevars"

        # Reset prompt
        if [ -n "${GTWS_SAVEPS1}" ]; then
                PS1="${GTWS_SAVEPS1}"
        fi
        if [ -n "${GTWS_SAVEPATH}" ]; then
                export PATH=${GTWS_SAVEPATH}
        fi
        unset LD_LIBRARY_PATH
        unset PYTHONPATH
        unset PROMPT_COMMAND
        unset CDPATH
        unset SDIRS

        # Save variables
        for i in ${savevars}; do
                SRC=GTWS_${i}
                DST=SAVE_${i}
                debug_print "\t $i: ${DST} = ${!SRC}"
                eval ${DST}=${!SRC}
        done

        # Clear GTWS evironment
        for i in ${!GTWS*} ; do
                if [ -n "${verbose}" ]; then
                        echo -e "unset $i" >&2
                fi
                unset $i
        done

        # Restore variables
        for i in ${savevars}; do
                SRC=SAVE_${i}
                DST=GTWS_${i}
                if [ -n "${verbose}" ]; then
                        echo -e "\t $i: ${DST} = ${!SRC}" >&2
                fi
                if [ -n "${!SRC}" ]; then
                        eval export ${DST}=${!SRC}
                fi
                unset ${SRC}
        done
}

# save_env ${file} ${nukevars}
#
# Save the environment of GTWS_* to the give file, except for the variables
# given to nuke.  The default values to nuke are given below.
function save_env {
        local fname=${1}
        local nukevars=${2:-"SAVEPATH ORIGIN WS_GUARD LOC SAVEPS1"}
        debug_print "nukevars=$nukevars"

        for i in ${!GTWS*} ; do
                for j in ${nukevars}; do
                        if [ "${i}" == "GTWS_${j}" ]; then
                                debug_print "skipping $i"
                                continue 2
                        fi
                done
                debug_print "saving $i"
                echo "export $i=\"${!i}\"" >> "${fname}"
        done
```

```
}

# gtws_tmux_session_name ${PROJECT} ${VERSION} ${WSNAME} sesname
#
# Result will be in local variable sesname  Or:
#
# sesname = $(gtws_tmux_session_name ${PROJECT} ${VERSION} ${WSNAME})
#
# Result will be in local variable sesname
#
# Get the tmux session name for a given workspace
function gtws_tmux_session_name {
        local project=$1
        local version=$2
        local wsname=$3
        local __resultvar=$4
        local sesname="${project//./_}/${version//./_}/${wsname//./_}"

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$sesname'"
        else
                echo "$sesname"
        fi
}

# gtws_tmux_session_info ${SESSION_NAME} running attached
#
# Determine if a session is running, and if it is attached
#
# Result will be in local variables running and attached
#
# Test with:
# if $running ; then
#       echo "is running"
# fi

function gtws_tmux_session_info {
        local ses_name=$1
        local __result_running=$2
        local __result_attached=$3

        local __num_ses=$(tmux ls | grep "^${ses_name}" | wc -l)
        local __attached=$(tmux ls | grep "^${ses_name}" | grep attached)

        echo "$ses_name ses=${__num_ses}"

        if [[ "$__result_running" ]]; then
                if [ "${__num_ses}" != "0" ]; then
                        eval $__result_running="true"
                else
                        eval $__result_running="false"
                fi
        fi
        if [[ "$__result_attached" ]]; then
                if [ -n "${__attached}" ]; then
                        eval $__result_attached="true"
                else
                        eval $__result_attached="false"
                fi
        fi
}

# gtws_tmux_kill ${BASENAME}
#
# Kill all sessiont matching a pattern
function gtws_tmux_kill {
        local basename=$1
        local old_sessions=$(tmux ls 2>/dev/null | fgrep "${basename}" | cut -f 1 -d:)
        for session in ${old_sessions}; do
                tmux kill-session -t "${session}"
```

```
        done
}

# gtws_tmux_cleanup
#
# Clean up defunct tmux sessions
function gtws_tmux_cleanup {
        local old_sessions=$(tmux ls 2>/dev/null | egrep "^[0-9]{14}.*[0-9]+\)$" | cut -f 1 -d:)
        for session in ${old_sessions}; do
                tmux kill-session -t "${session}"
        done
}

# gtws_tmux_attach ${SESSION_NAME}
#
# Attach to a primary session.  It will remain after detaching.
function gtws_tmux_attach {
        local ses_name=$1

        tmux attach-session -t "${ses_name}"
}

# gtws_tmux_slave ${SESSION_NAME}
#
# Create a secondary session attached to the primary session.  It will exit it
# is detached.
function gtws_tmux_slave {
        local ses_name=$1

        # Session is is date and time to prevent conflict
        local session=`date +%Y%m%d%H%M%S`
        # Create a new session (without attaching it) and link to base session
        # to share windows
        tmux new-session -d -t "${ses_name}" -s "${session}"
        # Attach to the new session
        gtws_tmux_attach "${session}"
        # When we detach from it, kill the session
        tmux kill-session -t "${session}"
}

function cdorigin() {
        if [ -n "$(declare -F | grep "gtws_project_cdorigin")" ]; then
                gtws_project_cdorigin $@
        else
                gtws_cdorigin $@
        fi
}

function gtws_get_origin {
        local opv=$1
        local target=$2
        local __origin=
        local __resultvar=$3

        # If it's a git repo with a local origin, use that.
        __origin=$(git config --get remote.origin.url)
        if [ ! -d "${__origin}" ]; then
                __origin="${__origin}.git"
        fi
        if [ ! -d "${__origin}" ]; then
                # Try to figure it out
                if [ ! -d "${opv}" ]; then
                        die "No opv for $target" || return 1
                fi
                find_git_repo "${opv}" "${target}" __origin || return 1
        fi

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$__origin'"
        else
```

```
                echo "$__origin"
        fi
}

function gtws_cdorigin() {
        local opv=$(gtws_opvn "${GTWS_ORIGIN}" "${GTWS_PROJECT}" "${GTWS_PROJECT_VERSION}" "${GTWS_WSNAME}")
        local gitdir=""
        local target=""
        if [ -n "$1" ]; then
                target="$@"
        else
                git_top_dir gitdir || return 1
                target=$(basename $gitdir)
        fi

        gtws_get_origin $opv $target origin || return 1
        cd "${origin}"
}

# Copy files to another machine in the same workspace
function wsrcp {
        local target="${!#}"
        local length=$(($#-1))
        local base=${PWD}

        if [ -z "${1}" -o -z "${2}" ]; then
                echo "usage: ${FUNCNAME} <path> [<path>...] <target>"
                return 1
        fi

        for path in "${@:1:$length}"; do
                gtws_rcp "${path}" "${target}:${base}/${path}"
        done
}

# Override "cd" inside the workspace to go to GTWS_WSPATH by default
function cd {
        if [ -z "$@" ]; then
                cd "${GTWS_WSPATH}"
        else
                builtin cd $@
        fi
}

# Generate diffs/interdiffs for changes and ship to WS on other boxes
function gtws_interdiff {
        local targets=$@
        local target=
        local savedir=${PWD}
        local topdir=$(git_top_dir)
        local repo=$(basename ${topdir})
        local mainpatch="${GTWS_WSPATH}/patches/${repo}-full.patch"
        local interpatch="${GTWS_WSPATH}/patches/${repo}-incremental.patch"

        if [ -z "${targets}" ]; then
                echo "Usage: ${FUNCNAME} <targethost>"
                  die "Must give targethost" || return 1
        fi
        cd "${topdir}"
        if [ -f "${mainpatch}" ]; then
                git diff | interdiff "${mainpatch}" - > "${interpatch}"
        fi
        git diff > "${mainpatch}"
        for target in ${targets}; do
                gtws_rcp "${mainpatch}" "${interpatch}" \
                        "${target}:${GTWS_WSPATH}/patches"
        done
        cd "${savedir}"
}
```

```
function gtws_debug {
        local cmd=$1
        if [ -z "${cmd}" ]; then
                echo "Must give a command"
                echo
                die "${FUNCNAME} <cmd-path>" || return 1
        fi
        local cmdbase=$(basename $cmd)
        local pid=$(pgrep "${cmdbase}")

        ASAN_OPTIONS="abort_on_error=1" cgdb ${cmd} ${pid}
}

# remote_cmd "${target}" "${command}" output
#
# Result will be in local variable output  Or:
#
# output = $(remote_cmd "${target}" "${command}")
#
# Result will be in local variable output
#
# Run a command remotely and capture sdtout.  Make sure to quote the command
# appropriately.
remote_cmd() {
        local target=$1
        local cmd=$2
        local __resultvar=$3
        local output=


        if [ -z "${GTWS_VERBOSE}" ]; then
                output=$(ssh "${target}" "${cmd}" 2>/dev/null)
        else
                output=$(ssh "${target}" "${cmd}")
        fi
        local ret=$?

        if [[ "$__resultvar" ]]; then
                eval $__resultvar="'$output'"
        else
                echo "${output}"
        fi
        return ${ret}
}
```