Manage Git repositories with a meta directory

Projects Everywhere

How do you make sure the new laptop you just bought is populated with copies of all the Git repositories you use? Easy. By using a meta repository to maintain a list of projects and Perl scripts to automate discovery and cloning. *By Mike Schilli*

The Git version control system easily outperforms old-timers such as CVS, Subversion, or Perforce. Having the entire project's history available offline and Git's branch strategy are such powerful features that many programmers wonder how they ever managed to develop software before the invention of decentralized version control systems.

Repository Collections

That said, Git typically focuses on a single project, and its support for subprojects is rudimentary at best. Because of this, active developers tend to create or clone dozens of Git repositories in the course of time, and their authoritative copies often reside on different servers. Keeping your local copies up to date then becomes a pain as the number of repositories grows, and all hell breaks loose if you happen to change machines and have to rediscover and reclone everything all at once. After buying a new laptop, or moving to a new development desktop, it would be really useful to have a copy of all your projects waiting for you.

Additionally, in many cases, computers will be assigned to groups that require different repositories. For example, you might not want to keep a Git repository with large images on your laptop for space reasons, and you would probably want to avoid storing private content on your computer at work. A configuration file, stored somewhere on the Internet, could store the locations of the repositories you use. These values tend to change quickly as you add new projects and delete or move others. This scenario sounds like a task for a version control system: How about using Git to maintain your metadata (meta) repository?

Invented Format

The configuration data will be stored in a plain text file in YAML format, because

MIKE SCHILLI

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.

```
# gitmeta.gmf

# Private Project via git/SSH
- username@private.server.com:git/private-project.git

# Perl core
- git://perl5.git.perl.org/perl.git

# All github projects of user 'mschilli'
-
    type: Github
    user: mschilli

# All projects in directory 'projects'
# on some host via git/SSH
-
    type: SshDir
    host: username@hoster.com
    dir:  projects
~
"gitmeta.gmf" 19L, 364C                1,1           All
```

Figure 1: The metadata to locate all of the user's active Git repositories are stored in the `gitmeta.gmf` file in the Git meta repository somewhere on the Internet.

it's easily read by both humans and machines. This specific new dialect shall be called GMF (Git Meta Format), and my configuration files will have a `.gmf` file extension.

Figure 1 gives an example. The first entry points to a privately hosted repository that resides on a fictitious server, *private.server.com*, which supports SSH access.

The second entry points to the official Git repository for the Perl 5 kernel, which stores the entire commit history ever since Larry Wall released the first version of Perl back in 1987. Both repository locators in the configuration can be used directly by the `git clone` command, which creates a local directory with a copy of the remote repository.

Of course, the GMF file could simply list all the active repositories, but busy developers would find the effort of continually adding new projects and removing defunct ones strenuous in the long term.

For example, if you launch a dozen projects on Github.com, or on a server with SSH access, you could save yourself the trouble of adding these locations if the meta repository understood how to interpret these collections without human interaction. The meta repository would thus need to understand instructions such as "Grab all the repositories in this directory on this server over SSH" or "All repositories by this user stored on GitHub."

In One Fell Swoop

The YAML blocks assigned to the two lower dashes in Figure 1 each represent two hashes (see Figure 2 for the Perl format to which the YAML configuration is converted) that designate collections of repositories with specific properties in the meta format.

The first hash has a value of `Github` in its `type` field, and the `user` entry, with its value of `mschilli`, indicates that all repositories belonging to user `mschilli` on Github.com should be copied to the local machine or updated.

Instead of dozens of separate entries, there are just two lines, and if the user were to create some new repositories on Github.com, they would automatically become part of the meta repository without requiring modification of the configuration file.

If the user deleted a project on GitHub, the updater would not explicitly delete the local project. But if the user removed the local copy, cloning would no longer occur.

The YAML entry next to the last dash in Figure 1 (or the final data structure in Figure 2) references a collection of Git

```
[
  'username@private.server.com:git/private-project.git',

  'git://perl5.git.perl.org/perl.git',

  { user => 'mschilli',
    type => 'Github'
  },

  { type => 'SshDir',
    dir  => 'projects',
    host => 'username@hoster.com'
  }
];
                                          1,1           All
```

Figure 2: The data parsed from the `gitmeta.gmf` YAML file gets transformed into a Perl data structure.

LISTING 1: gitmeta-update

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use GitMeta::GMF;
04 use Sysadm::Install qw(:all);
05 use File::Basename;
06 use Getopt::Std;
07 use Log::Log4perl qw(:easy);
08
09 getopts("vn", \my %opts);
10
11 if ($opts{v}) {
12  Log::Log4perl->easy_init(
13    $DEBUG);
14 }
15
16 my ($gmf_repo, $gmf_path,
17  $local_dir)
18    = @ARGV;
19
20 die "usage: $0 gmf-repo ",
21     "gmf-path local-dir"
22   unless defined $local_dir;
23
24 main();
25
26 ############################
27 sub main {
28 ############################
29  my $gm = GitMeta::GMF->new(
30    repo      => $gmf_repo,
31    gmf_path => $gmf_path
32  );
33
34  my @urls = $gm->expand();
35
36  if ($opts{n}) {
37    for my $url (@urls) {
38      print "$url\n";
39    }
40    return 1;
41  }
42
43  cd $local_dir;
44
45  for my $url (@urls) {
46    my $repo_dir =
47      basename $url;
48    $repo_dir =~ s/\.git$//g;
49    if (-d $repo_dir) {
50      cd $repo_dir;
51      tap "git", "fetch",
52        "origin";
53      cdback;
54    } else {
55      tap "git", "clone", $url;
56    }
57  }
58  return 1;
59 }
```

LISTING 2: GitMeta.pm

```
01 ############################
02 package GitMeta;
03 ############################
04
05 ############################
06 sub new {
07 ############################
08  my ($class, %options) = @_;
09
10  my $self = {%options};
11  bless $self, $class;
12 }
13
14 ############################
15 sub expand {
16 ############################
17  die "You need to ",
18      "implement 'expand'";
19 }
20
21 ############################
22 sub param_check {
23 ############################
24  my ($self, @params) = @_;
25
26  for my $param (@param) {
27   if (
28    !exists $self->{$param})
29   {
30    die "Parameter $param ",
31        " missing";
32   }
33  }
34 }
35
36 1;
```

repositories that reside in a directory on the specified server with SSH access. Again, the updater automatically picks up new entries without needing user interaction: To do this, the processing script lists the subdirectories and then clones the individual repositories it finds in this way.

Mirror, Mirror

The `gitmeta-update` script in Listing 1 handles the original cloning and later updating procedures of local repositories based on the data stored in the meta repository. The meta repository will typically reside on a server with SSH access to restrict the use of this potentially confidential meta information to the authorized user.

The script expects three command-line parameters: the location of the meta repository, the path to the GMF file within it, and the local directory in which the mirrored repositories will be stored. The following command line

```
gitmeta-update -v ➥
   user@secret.server.com:git/gitmeta ➥
   gitmeta.gmf ➥
   /path/to/local/repo/directory
```

contacts the server at *secret.server.com*, logs in via SSH as `user`, changes to the remote `git/gitmeta` directory below the home directory belonging to `user` and mirrors the Git repository it finds there to a temporary directory on the local disk. Then it loads the current version of the `gitmeta.gmf` file, runs it through the YAML parser, and processes the array entries one after another.

Incidentally, the `-v` option in the previous command line sends verbose output from the commands being processed to stderr, with some help from the Log4perl API.

LISTING 3: GMF.pm

```
01 ############################
02 package GitMeta::GMF;
03 ############################
04 use strict;
05 use warnings;
06 use base qw(GitMeta);
07 use File::Temp qw(tempdir);
08 use Log::Log4perl qw(:easy);
09 use YAML qw(Load);
10 use Sysadm::Install qw(:all);
11 use File::Basename;
12
13 ############################
14 sub expand {
15 ############################
16  my ($self) = @_;
17
18  $self->param_check("repo",
19   "gmf_path");
20
21  my $yml =
22    $self->_fetch(
23   $self->{repo},
24   $self->{gmf_path});
25
26  my @locs = ();
27
28  for my $entry (@$yml) {
29   my $type = ref($entry);
30
31   if ($type eq "") {
32
33    # plain git url
34    push @locs, $entry;
35   } else {
36    my $class =
37      "GitMeta::"
38      . ucfirst(
39     $entry->{type});
40    eval "require $class;"
41      or LOGDIE
42      "Class $class missing";
43    my $expander =
44      $class->new(%$entry);
45    push @locs,
46      $expander->expand();
47   }
48  }
49
50  return @locs;
51 }
52
53 ############################
54 sub _fetch {
55 ############################
56  my ($self, $git_repo,
57   $gmf_path)
58    = @_;
59
60  my ($tempdir) =
61    tempdir(CLEANUP => 1);
62
63  cd $tempdir;
64  tap "git", "clone",
65    $git_repo;
66  my $data =
67    slurp(basename($git_repo)
68     . "/$gmf_path");
69  cdback;
70  my $yml = Load($data);
71  return $yml;
72 }
73
74 1;
```

The Perl code wraps the retrieval and processing of the YAML file in the GitMeta::GMF class, but more of that later. Lines 29 through 32 in Listing 1 call the `new()` constructor and pass in the repository locator, `$gmf_repo`, along with the path `$gmf_path` to the GMF file. The call to the `expand()` method in line 34 resolves direct and indirect references in the YAML file and returns a list of repository locators that point to the repositories that need to be mirrored.

If the `-n` option is set, the script performs a dry run, and line 36 branches off to a `for` loop that only outputs the identified locators for test purposes and then terminates without actually mirroring anything. In production use, line 43 would use the Sysadm::Install module's `cd` command to change to the local mirror directory and start cranking.

The `for` loop in lines 45-56 iterates over all the found repository locators, removes any `.git` extensions from the names, and checks whether the corresponding directory already exists (i.e., whether the repository has already been mirrored). If so, it uses the command `git fetch` to retrieve the changes that happened in the remote location. It does not merge them with the local git branch, like a call to `git pull` would, because this could cause conflicts that the user would painstakingly have to resolve. The `gitmeta-update` command aims to create a fast mirror while the Internet connection is up. Once you have retrieved the changes, you can always use Git to merge them offline.

Fresh Clones

If no local directory for the repository exists yet, `git clone` in line 55 of Listing 1 creates one and then fetches the data from the remote repository to create a full clone.

"You might not want to keep a Git repository on your laptop for space reasons, and you probably want to avoid storing private content on your computer at work."

The whole magic of the script is contained in the GitMeta::GMF class and its `expand()` method, which is called in line 34 and doesn't just fetch a GMF file but recursively interprets its entries.

Listing 3 implements the GitMeta::GMF class, which inherits from the `GitMeta.pm` base class in Listing 2. Its `expand()` method expects two parameters: the repository locator, `repo`, and the relative path to the remote GMF file, `gmf_path`. The almost virtual base class in Listing 2 provides the standard constructor, `new()`, which is inherited by derived classes. This saves typing and avoids code duplication.

Lazy Subclasses

Additionally, the `GitMeta.pm` base class defines the `param_check()` method called by the subclasses to check whether their constructors have been handed the parameters they expect. The method terminates the program if any of them are missing. All subclasses refer to their base class `GitMeta` by a `use base qw(GitMeta)` statement, as in line 6 of Listing 3, for example.

The instance of the `expand()` method defined in the base class (line 15, Listing 2) simply contains an instruction that terminates the program and is never executed if the subclass defines its own `expand()` method. The `die` instruction serves as a reminder to subclass programmers to implement this virtual base class method in the subclass.

The `_fetch()` method defined in lines 54-72 (Listing 3) clones the specified Gitmeta repository into a temporary directory and slurps the YAML data provided by the GMF file into a Perl structure, which it returns as a result. The underscore in the method name indicates that this is an internal, private method that

LISTING 4: Github.pm

```
01 ############################
02 package GitMeta::Github;
03 ############################
04 use strict;
05 use warnings;
06 use base qw(GitMeta);
07 use LWP::UserAgent;
08 use XML::Simple;
09
10 ############################
11 sub expand {
12 ############################
13  my ($self) = @_;
14
15  $self->param_check("user");
16
17  my $user  = $self->{user};
18  my @repos = ();
19
20  my $ua =
21    LWP::UserAgent->new();
22  my $resp = $ua->get(
23    "http://github.com" .
24    "/api/v1/xml/$user"
25  );
26
27  if ($resp->is_error) {
28   die "API fetch failed: ",
29     $resp->message();
30  }
31
32  my $xml = XMLin(
33   $resp->decoded_content());
34
35  my $by_repo =
36    $xml->{repositories}
37    ->{repository};
38
39  for
40    my $repo (keys %$by_repo)
41  {
42   push @repos,
43     "git\@github.com" .
44     ":$user/$repo.git";
45  }
46
47  return @repos;
48 }
49
50 1;
```

```
# Include another git Meta repo's repositories.
-
    type: GMF
    repo: user@devhost.com:git/gitmeta
    gmf_path: privdev.gmf

# ...
"dev.gmf" 7L, 136C                1,1           All
```

Figure 3: You can easily reference other Git meta repositories from a Git meta repository to create a hierarchical structure.

does not belong to the API exported by the class.

Polymorphic Expansion

The exported `expand()` method first calls `_fetch()` and then iterates in the `for` loop starting in line 28 over all the YAML array elements found in the GMF file. If these elements are normal repository locators without a `type` entry, line 34 appends them to the `@locs` array without any modification. However, if the current YAML element contains a structure with an entry in its `type` field, `GMF.pm` delegates processing to a subclass of the corresponding type.

Supported values for `type` are `github` and `sshdir`, which pass on processing of the entry to the subclasses GitMeta::Github and GitMeta::SshDir, respectively. To allow this to happen, the `eval` command in line 40 finds and loads the required class to the active program; line 44 calls the class's constructor `new()` with the remaining parameters found in the YAML entry.

Following polymorphic tradition, all subclasses have an `expand()` method that returns lists of repository locators. Values returned, regardless of which instance of `expand()` found them, are sent to the end of the `@locs` array and contribute to the total list of repositories in the `main` script.

All My GitHub Projects

If the script comes across a `github` entry in the `type` field when interpreting a GMF file, it activates GitMeta::Github (Listing 4). This class also inherits from the `GitMeta` base class and simply overwrites its `expand()` method by fetching the names of all repositories residing on GitHub for the defined user. To do so, it uses GitHub's simple XML API, which is freely available under the `/api/v1/xml/username` path on Github.com and doesn't even require you to register or submit a token. Calling `decoded_content()` on the response coming back from the web server ensures that project descriptions encoded in UTF8 will still return valid XML.

The XML returned by the web query is then grabbed by the `XMLin()` function of the CPAN XML::Simple module, which converts it into a deeply nested hash data structure. Line 35 dives into the hash using the `{repositories}->{repository}` key and receives a hash whose keys represent the repository names.

> "The configuration data are stored in a plain text file in YAML format …"

Lines 43 and 44 create a typical GitHub-style repository locator from the name. Local users will have read and write access, assuming they identify themselves correctly with a valid SSH key on Github.com.

No Prying Eyes

Listing 5 contains another specialized class. The Gitmeta::SshDir package defined there, which also inherits from `GitMeta`, is responsible for repositories that reside as subdirectories in a directory on a server with SSH-protected access. This is perfect for private repositories, because neither the content nor the names are published anywhere.

To parse a list of the directories available on the server and to pass it to the updater later, line 21 (Listing 5) uses the SSH protocol to run an `ls` command on the server, thus obtaining the subdirectories of the given path. The output is separated by newlines, because the Unix shell sends it this way.

The `while` loop in lines 27-32 creates a repository locator for the Git-via-SSH protocol from each line and appends it to the resulting `@repos` array, which the method then passes back to the caller as a long list.

Meta repositories can also reference other meta repositories, as in Figure 3. The `type` field in the entry shown here has a value of `GMF`; the processing code thus hands over responsibility to the GitMeta::GMF class, which in turn fetches the remote repository and obtains and processes its GMF file.

The script resolves entries recursively and creates a long list of repositories that need updating. With this information, programmers then can define groups of repositories and assign a tailored collection of repositories to each system by cleverly combining different meta repositories, with no need to reference repositories multiple times in multiple configurations. The configuration shown in Figure 3 precisely matches

```
gitmeta-update ➥
    user@devhost.com:git/gitmeta ➥
    privdev.gmf ...
```

apart from the fact that the command line is followed by the name of the directory that is supposed to receive the local clones. The GMF files in the meta repository can also be stored in subdirectories to improve the structure. I could imagine creating a meta repository with two GMF files, `priv/free.gmf` and `priv/commerce.gmf`, to separate free software from commercial software. To reference either one instead of *privdev.gmf*, you would simply adapt the `gmf_path` in the GMF configuration or the second `gitmeta-update` parameter on the command line.

Keys Replace Passwords

To avoid repeatedly typing your password for SSH access, create private/public key pairs and install the public parts on all SSH servers involved. Otherwise, when the servers request a password, you won't see the prompt because the `tap()` commands eat them, which leaves you wondering what's going on. GitHub doesn't support passwords for Git ac-

```
mybox> ssh user@some.host.com
some.host> cd repos
some.host> mkdir gitmeta
some.host> cd gitmeta
some.host> git init
Initialized empty Git repository in .git/
some.host> vi gitmeta.gmf
...
some.host> git commit -m "initial version" gitmeta.gmf
```

Figure 4: To create new GMF files, create a new Git meta repository on a server with SSH access, edit the GMF file, and run a commit.

cess, anyway, and requires users to deposit their public keys on the website.

Installation

To run the `gitmeta-update` script on a newly installed machine, install the Perl and the CPAN modules it and its supporting classes use. The four classes I have mentioned must be stored in the following directory tree on your filesystem below a path the script can find:

```
GitMeta.pm
GitMeta/GMF.pm
GitMeta/Github.pm
GitMeta/Sshdir.pm
```

To make new GMF files, create a new Git repository on a server with SSH access, edit the GMF file, and run a `commit` (Figure 4). After creating the meta repository on the server, access it via a locator, like *user@some.host.com*:*repodir*/gitmeta.

The cloning starts when you run `git-meta-update` with this locator and a local target directory. If you don't have a new laptop to experiment on, this gives you a perfect excuse to buy one. ■■■

INFO

[1] Listings for this article: *http://www.linuxpromagazine.com/Resources/Article-Code*

LISTING 5: SshDir.pm

```
01 ############################
02 package GitMeta::SshDir;
03 ############################
04 use strict;
05 use warnings;
06 use base qw(GitMeta);
07 use Sysadm::Install qw(:all);
08 use Log::Log4perl qw(:easy);
09
10 ############################
11 sub expand {
12 ############################
13  my ($self) = @_;
14
15  $self->param_check("host",
16   "dir");
17
18  INFO "Retrieving repos ",
19    "from $self->{host}";
20
21  my ($stdout) = tap "ssh",
22    $self->{host},
23    "ls", $self->{dir};
24
25  my @repos = ();
26
27  while ($stdout =~ /(.*)\n/g)
28  {
29   push @repos,
30     "$self->{host}:" .
31     "$self->{dir}/$1";
32  }
33
34  return @repos;
35 }
36
37 1;
```