goiardi Documentation

Release 0.11.7

Jeremy Bingham

Jan 27, 2018

Contents

Goiardi is an implementation of the Chef server (http://www.chef.io) written in Go. It can either run entirely in memory with the option to save and load the in-memory data and search indexes to and from disk, drawing inspiration from chef-zero, or it can use MySQL or PostgreSQL as its storage backend. Cookbooks can either be stored locally, or optionally in Amazon S3 (or a compatible service).

Like all software, it is a work in progress. Goiardi now, though, should have all the functionality of the open source Chef Server, plus some extras like reporting, event logging, and a Chef Push-like feature called "shovey". It does not support other Enterprise Chef type features like organizations at this time. When used, knife works, and chef-client runs complete successfully. Almost all chef-pendant tests successfully successfully run, with a few disagreements about error messages that don't impact the clients. It does pretty well against the official chef-pedant, but because goiardi handles some authentication matters a little differently than the official chef-server, there is also a fork of chef-pedant located at https://github.com/ctdk/chef-pedant that's more custom tailored to goiardi.

Many go tests are present as well in different goiardi subdirectories.

CHAPTER 1

Dependencies

As of version 0.11.0, goiardi now includes its dependencies in the `vendor` directory. This saves the headache of having to download various sources and possibly finding that they don't work.

If, for whatever reason, you are building goiardi with vendoring disabled, the dependencies will be installed when you `go get` it.

If you would like to modify the search grammar, you'll need the `peg` package. To install that, run:

```
go get github.com/pointlander/peg
```

In the `search/` directory, run `peg -switch -inline search-parse.peg` to generate the new grammar. If you don't plan on editing the search grammar, though, you won't need that.

CHAPTER 2

Installation

To install goiardi from source:

1. Install go. (http://golang.org/doc/install.html) Goiardi now requires go 1.7+ (because of the use of contexts). Goiardi should generally be able to be built with the latest version of Go, and this is generally recommended. Usually it will also build with the previous minor release, and may build with older versions as well, but this shouldn't be relied on. Immediately after a minor release, of course, caution may be warranted.
2. Make sure your `$GOROOT` and `$PATH` are set up correctly per the Go installation instructions.
3. Download goairdi and its dependencies

   ```
   go get -t -u github.com/ctdk/goiardi
   ```

4. Run tests, if desired. Several goiardi subdirectories have go tests, and chef-pedant can and should be used for testing goiardi as well.
5. Install the goiardi binaries.

   ```
   go install github.com/ctdk/goiardi
   ```

6. Run goiardi.

   ```
   goiardi <options>
   ```

   Or, you can look at the goiardi releases page on github at https://github.com/ctdk/goiardi/releases and see if there are precompiled binaries available for your platform, or check out the packages at https://packagecloud.io/ct/goiardi and see if there's one for your platform there.

Another option is running goiardi in Docker. There's a Dockerfile in the root of the goiardi git repository that's suitable for running the local version of goiardi, but a goiardi repository on Docker Hub at https://hub.docker.com/r/ctdk/goiardi/ is also under development (the source repository for those docker images is at https://github.com/ctdk/goiardi-docker). Running goiardi under docker has always worked fine, but now that configuration options can be set with environment variables it's certainly easier to do so than before.

CHAPTER 3

Configuration

You can get a list of command-line options with the `-h` flag.

Additionally, many of goiardi's options that can be set with flags can also be set with environment variables. Where this is the case, the option's description will be followed by an environment variable name (like `$GOIARDI_HANDY_OPTION`).

Goiardi can also take a config file, run like `goiardi -c /path/to/conf-file`. See `etc/goiardi.conf-sample` for an example documented configuration file. Options in the configuration file share the same name as the long command line arguments (so, for example, `--ipaddress=127.0.0.1` on the command line would be `ipaddress = "127.0.0.1"` in the config file.

Currently available command line and config file options:

```
  -v, --version               Print version info.
  -V, --verbose               Show verbose debug information. Repeat for more
                              verbosity.
  -c, --config=               Specify a config file to use. [$GOIARDI_CONFIG]
  -I, --ipaddress=            Listen on a specific IP address.
                              [$GOIARDI_IPADDRESS]
  -H, --hostname=             Hostname to use for this server. Defaults to
                              hostname reported by the kernel.
                              [$GOIARDI_HOSTNAME]
  -P, --port=                 Port to listen on. If port is set to 443, SSL
                              will be activated. (default: 4545) [$GOIARDI_PORT]
  -Z, --proxy-hostname=       Hostname to report to clients if this goiardi
                              server is behind a proxy using a different
                              hostname. See also --proxy-port. Can be used with
                              --proxy-port or alone, or not at all.
                              [$GOIARDI_PROXY_HOSTNAME]
  -W, --proxy-port=           Port to report to clients if this goiardi server
                              is behind a proxy using a different port than the
                              port goiardi is listening on. Can be used with
                              --proxy-hostname or alone, or not at all.
                              [$GOIARDI_PROXY_PORT]
  -i, --index-file=           File to save search index data to.
                              [$GOIARDI_INDEX_FILE]
```

```
  -D, --data-file=            File to save data store data to.
                              [$GOIARDI_DATA_FILE]
  -F, --freeze-interval=      Interval in seconds to freeze in-memory data
                              structures to disk if there have been any changes
                              (requires -i/--index-file and -D/--data-file
                              options to be set). (Default 10 seconds.)
                              [$GOIARDI_FREEZE_INTERVAL]
  -L, --log-file=             Log to file X [$GOIARDI_LOG_FILE]
  -s, --syslog                Log to syslog rather than a log file.
                              Incompatible with -L/--log-file. [$GOIARDI_SYSLOG]
  -g, --log-level=            Specify logging verbosity. Performs the same
                              function as -V, but works like the 'log-level'
                              option in the configuration file. Acceptable
                              values are 'debug', 'info', 'warning', 'error',
                              'critical', and 'fatal'. [$GOIARDI_LOG_LEVEL]
      --time-slew=            Time difference allowed between the server's
                              clock and the time in the X-OPS-TIMESTAMP header.
                              Formatted like 5m, 150s, etc. Defaults to 15m.
                              [$GOIARDI_TIME_SLEW]
      --conf-root=            Root directory for configs and certificates.
                              Default: the directory the config file is in, or
                              the current directory if no config file is set.
                              [$GOIARDI_CONF_ROOT]
  -A, --use-auth              Use authentication. Default: false. (NB: At a
                              future time, the default behavior will change to
                              authentication being enabled.) [$GOIARDI_USE_AUTH]
      --use-ssl               Use SSL for connections. If --port is set to 433,
                              this will automatically be turned on. If it is
                              set to 80, it will automatically be turned off.
                              Default: off. Requires --ssl-cert and --ssl-key.
                              [$GOIARDI_USE_SSL]
      --ssl-cert=             SSL certificate file. If a relative path, will be
                              set relative to --conf-root. [$GOIARDI_SSL_CERT]
      --ssl-key=              SSL key file. If a relative path, will be set
                              relative to --conf-root. [$GOIARDI_SSL_KEY]
      --https-urls            Use 'https://' in URLs to server resources if
                              goiardi is not using SSL for its connections.
                              Useful when goiardi is sitting behind a reverse
                              proxy that uses SSL, but is communicating with
                              the proxy over HTTP. [$GOIARDI_HTTPS_URLS]
      --disable-webui         If enabled, disables connections and logins to
                              goiardi over the webui interface.
                              [$GOIARDI_DISABLE_WEBUI]
      --use-mysql             Use a MySQL database for data storage. Configure
                              database options in the config file.
                              [$GOIARDI_USE_MYSQL]
      --use-postgresql        Use a PostgreSQL database for data storage.
                              Configure database options in the config file.
                              [$GOIARDI_USE_POSTGRESQL]
      --local-filestore-dir=  Directory to save uploaded files in. Optional
                              when running in in-memory mode, *mandatory*
                              (unless using S3 uploads) for SQL mode.
                              [$GOIARDI_LOCAL_FILESTORE_DIR]
      --log-events            Log changes to chef objects. [$GOIARDI_LOG_EVENTS]
  -K, --log-event-keep=       Number of events to keep in the event log. If
                              set, the event log will be checked periodically
                              and pruned to this number of entries.
                              [$GOIARDI_LOG_EVENT_KEEP]
```

```
    --skip-log-extended     If set, do not save a JSON encoded blob of the
                            object being logged when logging an event.
                            [$GOIARDI_SKIP_LOG_EXTENDED]
-x, --export=               Export all server data to the given file, exiting
                            afterwards. Should be used with caution. Cannot
                            be used at the same time as -m/--import.
-m, --import=               Import data from the given file, exiting
                            afterwards. Cannot be used at the same time as
                            -x/--export.
-Q, --obj-max-size=         Maximum object size in bytes for the file store.
                            Default 10485760 bytes (10MB).
                            [$GOIARDI_OBJ_MAX_SIZE]
-j, --json-req-max-size=    Maximum size for a JSON request from the client.
                            Per chef-pedant, default is 1000000.
                            [$GOIARDI_JSON_REQ_MAX_SIZE]
    --use-unsafe-mem-store  Use the faster, but less safe, old method of
                            storing data in the in-memory data store with
                            pointers, rather than encoding the data with gob
                            and giving a new copy of the object to each
                            requestor. If this is enabled goiardi will run
                            faster in in-memory mode, but one goroutine could
                            change an object while it's being used by
                            another. Has no effect when using an SQL backend.
                            (DEPRECATED - will be removed in a future
                            release.)
    --db-pool-size=         Number of idle db connections to maintain. Only
                            useful when using one of the SQL backends.
                            Default is 0 - no idle connections retained
                            [$GOIARDI_DB_POOL_SIZE]
    --max-connections=      Maximum number of connections allowed for the
                            database. Only useful when using one of the SQL
                            backends. Default is 0 - unlimited.
                            [$GOIARDI_MAX_CONN]
    --use-serf              If set, have goidari use serf to send and receive
                            events and queries from a serf cluster. Required
                            for shovey. [$GOIARDI_USE_SERF]
    --serf-event-announce   Announce log events and joining the serf cluster
                            over serf, as serf events. Requires --use-serf.
                            [$GOIARDI_SERF_EVENT_ANNOUNCE]
    --serf-addr=            IP address and port to use for RPC communication
                            with a serf agent. Defaults to 127.0.0.1:7373.
                            [$GOIARDI_SERF_ADDR]
    --use-shovey            Enable using shovey for sending jobs to nodes.
                            Requires --use-serf. [$GOIARDI_USE_SHOVEY]
    --sign-priv-key=        Path to RSA private key used to sign shovey
                            requests. [$GOIARDI_SIGN_PRIV_KEY]
    --dot-search            If set, searches will use . to separate elements
                            instead of _. [$GOIARDI_DOT_SEARCH]
    --convert-search        If set, convert _ syntax searches to . syntax.
                            Only useful if --dot-search is set.
                            [$GOIARDI_CONVERT_SEARCH]
    --pg-search             Use the new Postgres based search engine instead
                            of the default ersatz Solr. Requires
                            --use-postgresql, automatically turns on
                            --dot-search. --convert-search is recommended,
                            but not required. [$GOIARDI_PG_SEARCH]
    --use-statsd            Whether or not to collect statistics about
                            goiardi and send them to statsd.
```

```
                              [$GOIARDI_USE_STATSD]
      --statsd-addr=          IP address and port of statsd instance to connect
                              to. (default 'localhost:8125')
                              [$GOIARDI_STATSD_ADDR]
      --statsd-type=          statsd format, can be either 'standard' or
                              'datadog' (default 'standard')
                              [$GOIARDI_STATSD_TYPE]
      --statsd-instance=      Statsd instance name to use for this server.
                              Defaults to the server's hostname, with '.'
                              replaced by '_'. [$GOIARDI_STATSD_INSTANCE]
      --use-s3-upload         Store cookbook files in S3 rather than locally in
                              memory or on disk. This or --local-filestore-dir
                              must be set in SQL mode. Cannot be used with
                              in-memory mode. [$GOIARDI_USE_S3_UPLOAD]
      --aws-region=           AWS region to use S3 uploads.
                              [$GOIARDI_AWS_REGION]
      --s3-bucket=            The name of the S3 bucket storing the files.
                              [$GOIARDI_S3_BUCKET]
      --aws-disable-ssl       Set to disable SSL for the endpoint. Mostly
                              useful just for testing.
                              [$GOIARDI_AWS_DISABLE_SSL]
      --s3-endpoint=          Set a different endpoint than the default
                              s3.amazonaws.com. Mostly useful for testing with
                              a fake S3 service, or if using an S3-compatible
                              service. [$GOIARDI_S3_ENDPOINT]
      --s3-file-period=       Length of time, in minutes, to allow files to be
                              saved to or retrieved from S3 by the client.
                              Defaults to 15 minutes. [$GOIARDI_S3_FILE_PERIOD]
      --use-external-secrets  Use an external service to store secrets
                              (currently user/client public keys). Currently
                              only vault is supported.
                              [$GOIARDI_USE_EXTERNAL_SECRETS]
      --vault-addr=           Specify address of vault server (i.e.
                              https://127.0.0.1:8200). Defaults to the value of
                              VAULT_ADDR.
      --vault-shovey-key=     Specify a path in vault holding shovey's private
                              key. The key must be put in vault as
                              'privateKey=<contents>'.
                              [$GOIARDI_VAULT_SHOVEY_KEY]
  -T, --index-val-trim=       Trim values indexed for chef search to this many
                              characters (keys are untouched). If not set or
                              set <= 0, trimming is disabled. This behavior
                              will change with the next major release.
                              [$GOIARDI_INDEX_VAL_TRIM]
  -y, --pprof-whitelist=      Address to allow to access /debug/pprof (in
                              addition to localhost). Specify multiple times to
                              allow more addresses. [$GOIARDI_PPROF_WHITELIST]
      --purge-reports-after=  Time to purge old reports after, given in golang
                              duration format (e.g. "720h"). Default is not to
                              purge them at all. [$GOIARDI_PURGE_REPORTS_AFTER]
      --purge-status-after=   Time to purge old node statuses after, given in
                              golang duration format (e.g. "720h"). Default is
                              not to purge them at all.
                              [$GOIARDI_PURGE_STATUS_AFTER]

MySQL connection options (requires --use-mysql):
      --mysql-username=       MySQL username [$GOIARDI_MYSQL_USERNAME]
      --mysql-password=       MySQL password [$GOIARDI_MYSQL_PASSWORD]
```

```
      --mysql-protocol=       MySQL protocol (tcp or unix)
                              [$GOIARDI_MYSQL_PROTOCOL]
      --mysql-address=        MySQL IP address, hostname, or path to a socket
                              [$GOIARDI_MYSQL_ADDRESS]
      --mysql-port=           MySQL TCP port [$GOIARDI_MYSQL_PORT]
      --mysql-dbname=         MySQL database name [$GOIARDI_MYSQL_DBNAME]
      --mysql-extra-params=   Extra configuration parameters for MySQL. Specify
                              them like '--mysql-extra-params=foo:bar'.
                              Multiple extra parameters can be specified by
                              supplying the --mysql-extra-params flag multiple
                              times. If using an environment variable, split up
                              multiple parameters with #, like so:
                              GOIARDI_MYSQL_EXTRA_PARAMS='foo:bar#baz:bug'.
                              [$GOIARDI_MYSQL_EXTRA_PARAMS]

PostgreSQL connection options (requires --use-postgresql):
      --postgresql-username=  PostgreSQL user name
                              [$GOIARDI_POSTGRESQL_USERNAME]
      --postgresql-password=  PostgreSQL password [$GOIARDI_POSTGRESQL_PASSWORD]
      --postgresql-host=      PostgreSQL IP host, hostname, or path to a socket
                              [$GOIARDI_POSTGRESQL_HOST]
      --postgresql-port=      PostgreSQL TCP port [$GOIARDI_POSTGRESQL_PORT]
      --postgresql-dbname=    PostgreSQL database name
                              [$GOIARDI_POSTGRESQL_DBNAME]
      --postgresql-ssl-mode=  PostgreSQL SSL mode ('enable' or 'disable')
                              [$GOIARDI_POSTGRESQL_SSL_MODE]
```

NB: If goiardi has been compiled with the `novault` build tag, the help output will be missing `--use-external-secrets`, `--vault-addr`, and `--vault-shovey-key`.

Options specified on the command line override options in the config file. Options specified via the command line override options in the config file, but are themselves overridden by command line flags.

For more documentation on Chef, see http://docs.chef.io.

CHAPTER 4

Binaries and Packages

There are other options for installing goiardi, in case you don't want to build it from scratch. Binaries for several platforms are provided with each release, and there are .debs available as well at https://packagecloud.io/ct/goiardi. At the moment packages are being built for Debian wheezy and later, Ubuntu 14.04 and later current and upcoming releases, raspbian (which is under the Debian versions) for various Raspberry Pi computers, and CentOS 6 and 7. Packages for other platforms may happen down the road. As of this writing, debs for goiardi 0.11.2 can be found in Debian stretch (a.k.a stable). More current versions of goiardi can be found in Debian's `testing` and `unstable` branches as well as in Ubuntu's `universe` repository since "Zesty Zapus".

NB: *wheezy* is currently (as of this writing) supported by the Debian LTS project. Sometime after that ends, which is scheduled for May 31st, 2018, it'll be dropped from the packagecloud.io builds and the supporting files removed from the repository.

There is also a homebrew tap that includes goiardi now, for folks running Mac OS X and using homebrew.

CHAPTER 5

Upgrading

Upgrading goiardi is generally a straightforward process. Usually all you should need to do is get the new sources and rebuild (using the `-u` flag when running `go get` to update goiardi is a good idea to ensure the dependencies are up to date), or download the appropriate new binary. However, sometimes a little more work is involved. Check the release notes for the new release in question for any extra steps that may need to be done. If you're running one of the SQL backends, you may need to apply database patches (either with sqitch or by hand), and in-memory mode especially may require using the data import/export functionality to dump and load your chef data between upgrades if the binary save file compatibility breaks between releases. However, while it should not happen often, occasionally more serious preparation will be needed before upgrading. It won't happen without a good reason, and the needed steps will be clearly outlined to make the process as painless as possible.

As a special note, if you are upgrading from any release prior to 0.6.1-pre1 to 0.7.0 and are using one of the SQL backends, the upgrade is one of the special cases. Between those releases the way the complex data structures associated with cookbook versions, nodes, etc. changed from using gob encoding to json encoding. It turns out that while gob encoding is indeed faster than json (and was in all the tests I had thrown at it) in the usual case, in this case json is actually significantly faster, at least once there are a few thousand coobkooks in the database. In-memory datastore (including file-backed in-memory datastore) users are advised to dump and reload their data between upgrading from <= 0.6.1-pre1 and 0.7.0, but people using either MySQL or Postgres *have* to do these things:

- Export their goiardi server's data with the `-x` flag.
- Either revert all changes to the db with sqitch, then redeploy, or drop the database manually and recreate it from either the sqitch patches or the full table dump of the release (provided starting with 0.7.0)
- Reload the goiardi data with the `-m` flag.

It's a fairly quick process (a goiardi dump with the `-x` flag took 15 minutes or so to load with over 6200 cookbooks) at least, but if you don't do it very little of your goiardi environment will work correctly. The above steps will take care of it.

If you're upgrading from version 0.8.2 (or before) to version 0.9.0, you will need to remove the search index save file before starting the new goiardi for the first time. After that's been done, run `knife index rebuild` to rebuild the search index.

One thing that's not always necessary, but is often good practice when running the in-memory trie based index, is to rebuild the search index with `knife index rebuild`. If anything has changed with the search index between releases, even if it's a minor one not worth making a new minor point release, rebuilding can help avoid any potential

gotchas. Postgres index users should only need to reindex when it's specifically noted that they should, although rebuilding won't hurt in that case.

CHAPTER 6

Supported Platforms

Goiardi has been built and run with the native 6g compiler on Mac OS X (10.7 and above), Debian squeeze, wheezy, and jessie, a fairly recent Arch Linux, FreeBSD 9.2, Ubuntu 14.04, Solaris, and Raspbian (on both the original Raspberry Pi and the Raspberry Pi 2). Using Go's cross compiling capabilities, goiardi builds for all of Go's supported platforms except plan9 (because of issues with the postgres client library). Windows support has not been tested extensively, but a cross compiled binary has been tested successfully on Windows.

At one point goiardi was able to be built and run with gccgo (using the `-compiler gccgo` option with the `go` command) on Arch Linux. Unfortunately while recent gccgo versions include the `go` command, so building go programs with gccgo is theoretically much easier than before, it currently doesn't actually work because some dependencies blow up under gccgo.

CHAPTER 7

Authentication

If goiardi is not running in use-auth mode, it does not actually care about .pem files at all. You still need to have one to keep knife and chef-client happy. It's like chef-zero in that regard.

If goiardi is running in use-auth mode, then proper keys are needed. When goiardi is started, if the chef-webui and chef-validator clients, and the admin user, are not present, it will create new keys in the `--conf-root` directory. Use them as you would normally for validating clients, performing tasks with the admin user, or using chef-webui if webui will run in front of goiardi.

In auth mode, goiardi supports versions 1.0, 1.1, and 1.2 of the Chef authentication protocol.

Note: The admin user, when created on startup, does not have a password. This prevents logging in to the webui with the admin user, so a password will have to be set for admin before doing so.

7.1 Fresh start

If you have not started the server without authentication and a persistent data store configured, just start it with authentication enabled and a conf-root directory. On the first start the admin, chef-webui, chef-validator keys will be saved to the directory given with the conf-root option.

7.2 Server saved data without authentication enabled

This means that the clients were created in the database but you don't have private keys for them. You need to start the server with authentication disabled, then use knife to regenerate the admin user's private key with `knife user reregister admin`. Save this key, and you can now enable authentication and use this key for the admin user. You'll have to recreate the chef-webui and chef-validator keys as well using a similar knife command (`knife client reregister <name>`), but you don't have to have authentication authentication disabled anymore, since you are authenticated with your new primary key.

CHAPTER 8

General Database Options

There are two general options that can be set for either database: `--db-pool-size` and `--max-connections` (and their configuration file equivalents `db-pool-size` and `max-connections`). `--db-pool-size` sets the number of idle connections to keep open to the database, and `--max-connections` sets the maximum number of connections to open on the database. If they are not set, the default behavior is to keep no idle connections alive and to have unlimited connections to the database.

It should go without saying that these options don't do much if you aren't using one of the SQL backends.

Of the two databases available, PostgreSQL is the better supported and recommended configuration. MySQL still works, of course, but it can't take advantage of some of the very helpful Postgres features.

8.1 MySQL mode

Goiardi can use MySQL to store its data, instead of keeping all its data in memory (and optionally freezing its data to disk for persistence).

If you want to use MySQL, you (unsurprisingly) need a MySQL installation that goiardi can access. This document assumes that you are able to install, configure, and run MySQL.

Once the MySQL server is set up to your satisfaction, you'll need to install sqitch to deploy the schema, and any changes to the database schema that may come along later. It can be installed out of CPAN or homebrew; see "Installation" on http://sqitch.org for details.

The sqitch MySQL tutorial at https://metacpan.org/pod/sqitchtutorial-mysql explains how to deploy, verify, and revert changes to the database with sqitch, but the basic steps to deploy the schema are:

- Create goiardi's database: `mysql -u root --execute 'CREATE DATABASE goiardi'`
- Optionally, create a separate mysql user for goiardi and give it permissions on that database.
- In sql-files/mysql-bundle, deploy the bundle: `sqitch deploy db:mysql://root[:<password>]@/goiardi`

To update an existing database deployed by sqitch, run the `sqitch deploy` command above again.

If you really really don't want to install sqitch, apply each SQL patch in sql-files/mysql-bundle by hand in the same order they're listed in the sqitch.plan file.

The above values are for illustration, of course; nothing requires goiardi's database to be named "goiardi". Just make sure the right database is specified in the config file.

Set `use-mysql = true` in the configuration file, or specify `--use-mysql` on the command line. If both the `-D`/`--data-file` flag and `--use-mysql` are used at the same time, an error will be printed to the log and the data file option will be ignored.

An example configuration is available in `etc/goiardi.conf-sample`, and is given below:

```
[mysql]
    username = "foo" # technically optional, although you probably want it
    password = "s3kr1t" # optional, if you have no password set for MySQL
    protocol = "tcp" # optional, but set to "unix" for connecting to MySQL
             # through a Unix socket.
    address = "localhost"
    port = "3306" # optional, defaults to 3306. Not used with sockets.
    dbname = "goiardi"
    # See https://github.com/go-sql-driver/mysql#parameters for an
    # explanation of available parameters
    [mysql.extra_params]
        tls = "false"
```

A similar example for configuring MySQL access via the command line is below:

```
goiardi -A --conf-root=/Users/jeremy/etc/goiardi --ipaddress="0.0.
0.0" --log-level="debug" --local-filestore-dir=/var/lib/goiardi/
lfs --db-pool-size=25 --use-mysql --mysql-username=goiardi
--mysql-address=localhost --mysql-extra-params=tls:false -i /var/goiardi/idx.
bin --mysql-dbname=goiardi
```

8.2 Postgres mode

Goiardi can also use Postgres as a backend for storing its data, instead of using MySQL or the in-memory data store. The overall procedure is pretty similar to setting up goiardi to use MySQL. Specifically for Postgres, you may want to create a database especially for goiardi, but it's not mandatory. If you do, you may also want to create a user for it. If you decide to do that:

- Create the user: `$ createuser goiardi <additional options>`
- Create the database, if you decided to: `$ createdb goiardi_db <additional options>`. If you created a user, make it the owner of the goiardi db with `-O goiardi`.

After you've done that, or decided to use an existing database and user, deploy the sqitch bundle in sql-files/postgres-bundle. If you're using the default Postgres user on the local machine, `sqitch deploy db:pg:<dbname>` will be sufficient. Otherwise, the deploy command will be something like `sqitch deploy db:pg://user:password@localhost/goairdi_db`.

The Postgres sqitch tutorial at https://metacpan.org/pod/sqitchtutorial explains more about how to use sqitch and Postgres.

Set `use-postgresql` in the configuration file, or specify `--use-postgresql` on the command line. Specifying both `-D`/`--data-file` flag and `--use-postgresql` at the same time will print an error to the log and ignore the data file setting, like how it works in MySQL mode. MySQL and Postgres cannot be used at the same time, also, and will result in a fatal error.

There is also an example Postgres configuration in the config file, and can be seen below:

```
# PostgreSQL options. If "use-postgres" is set to true on the command line or in
# the configuration file, connect to postgres with the options in [postgres].
# These options are all strings. See
# http://godoc.org/github.com/lib/pq#hdr-Connection_String_Parameters for details
# on the connection parameters. All of these parameters are technically optional,
# although chances are pretty good that you'd want to set at least some of them.
[postgresql]
    username = "foo"
    password = "s3kr1t"
    host = "localhost"
    port = "5432"
    dbname = "mydb"
    sslmode = "disable"
```

A command line flag sample would be something like this:

```
goiardi -A --conf-root=/etc/goiardi --ipaddress="0.0.0.0" --log-level="debug"
--local-filestore-dir=/var/lib/goiardi/lfs --pg-search --convert-search
--db-pool-size=25 --use-postgresql --postgresql-username=goiardi
--postgresql-host=localhost --postgresql-dbname=goiardidb
--postgresql-ssl-mode=disable
```

8.3 Note regarding goiardi persistence and freezing data

As mentioned above, goiardi can now freeze its in-memory data store and index to disk if specified. It will save before quitting if the program receives a SIGTERM or SIGINT signal, along with saving every "freeze-interval" seconds automatically if there have been any changes.

Saving automatically helps guard against the case where the server receives a signal that it can't handle and forces it to quit. In addition, goiardi will not replace the old save files until the new one is all finished writing. However, it's still not anywhere near a real database with transaction protection, etc., so while it should work fine in the general case, possibilities for data loss and corruption do exist. The appropriate caution is warranted.

CHAPTER 9

Import and Export of Data

Goiardi can now import and export its data in a JSON file. This can help both when upgrading, when the on-disk data format changes between releases, and to convert your goiardi installation from in-memory to MySQL (or vice versa). The JSON file has a version number set (currently 1.0), so that in the future if there is some sort of incompatible change to the JSON file format the importer will be able to handle it.

Before importing data, you should back up any existing data and index files (and take a snapshot of the SQL db, if applicable) if there's any reason you might want it around later. After exporting, you may wish to hold on to the old installation data until you're satisfied that the import went well.

Remember that the JSON export file contains the client and user public keys (which for the purposes of goiardi and chef are private) and the user hashed passwords and password salts. The export file should be guarded closely.

The `-x/--export` and `-m/--import` flags control importing and exporting data. To export data, stop goiardi, then run it again with the same options as before but adding `-x <filename>` to the command. This will export all the data to the given filename, and goiardi will exit.

Importing is ever so slightly trickier. You should remove any existing data store and index files, and if using an SQL database use sqitch to revert and deploy all of the SQL files to set up a completely clean schema for goiardi. Then run goiardi with the new options like you normally would, but add `-m <filename>`. Goiardi will run, import the new data, and exit. Assuming it went well, the data will be all imported. The export dump does not contain the user and client .pem files, so those will need to be saved and moved as needed.

Theoretically a properly crafted export file could be used to do bulk loading of data into goiardi, thus goiardi does not wipe out the existing data on its own but rather leaves that task to the administrator. This functionality is merely theoretical and completely untested. If you try it, you should back your data up first.

CHAPTER 10

Search

Goiardi currently has two different ways of running searches: the original and default ersatz Solr implementation, and a Postgres based search using ltree and trigrams with some new database tables. Both use the usual Solr syntax that chef expects, but are quite different under the hood.

Additional different search backends are now a possibility as well; goiardi search's archictecture has changed to make it easier to add new search backends, like actual Solr search or what have you.

10.1 Ersatz Solr Search

Nothing special needs to be done to use this search. It remains the default search implementation, and the only choice for the in-memory/file based storage and MySQL. It works well for smaller installations, but when you get in the neighborhood of hundreds of nodes it begins to get bogged down.

10.2 Postgres Search

Starting with goiardi version 0.10.0, there is an optional PostgreSQL based search. It uses the same solr parser that the default search backend uses, but instead of using tries to search for objects, it uses ltree and trigrams to search for values stored in a separate table. The postgres search is able to use the same solr query parser the original search uses to create postgres queries from the solr queries.

In testing, goiardi with postgres search can handle 10,000 nodes without any particular problem. Simple queries complete reasonably quickly, but more complex queries can take longer. In the most recent tests, on a 2014 MacBook Pro with 16GB of RAM and a totally untuned PostgreSQL installation, executing the search query equivalent to "data_center:Vagrantheim" directly into the database with 10,000 nodes consistently took about 40-60 milliseconds. The equivalent of "data_center:Vagrantheim AND name:server2*" took between 3 and 4 seconds, while "data_center:Vagrantheim AND name:(server2* OR server4*)" took about 7-8 seconds. It is expected that with proper tuning, and as this feature matures, these numbers will go down. It's also worth mentioning that when using knife search, the whole process takes considerably longer anyway.

The postgres search should be able to handle almost any query you throw at it, but it's definitely possible to craft a query that goiardi will fail to handle correctly. Particularly, if you're using fuzzy or distance searches, it will probably not return what you want. This postgres search should handle all normal cases, however.

The postgres based search still uses the same Solr syntax that chef search traditionally uses, but the Solr queries are parsed out and used to generate SQL queries for searching. There is likely room for improvement with the generated queries. An intriguing possibility for down the road is to allow an alternate query syntax that more closely reflects postgres' capabilities with these indexes.

The biggest issue between the standard Solr search with Chef and the goiardi Postgres based search is that ltree indices in Postgres can only use alphanumeric characters (plus _), with "." as a path separator. Since attributes can have whatever arbitrary characters you want in them, goiardi strips those characters out when they're indexed and when searching. This is not usually a problem, but could lead to strange results if you had something like "/dev/xvda1" AND "dev_xvda1" as attribute names in a node.

One difference that's worth mentioning is that you can start a search term with a wildcard character with the postgres search, unlike with the Solr searches.

To use the postgres search in your goiardi installation, you must:

1. be using postgres (duh) and
2. enable it in your goiardi.conf file with `pg-search = true`.

It is strongly recommended that you also set `convert-search = true`, because the postgres search uses the dot separator between path items instead of the underscore, and this will break existing search queries. If `index-file` is set, goiardi will print a warning that it's not very useful to have the index file enabled, but it's not a fatal error.

NB: It is also *very* strongly recommended, especially if you run chef-client frequently in a cron or as a daemon, that you periodically reindex the search_items table. Otherwise, the indexes can grow to ridiculous sizes and you'll be wondering why you're running out of space for no clear reason. The procedure is simple, however: add a command like `echo 'REINDEX TABLE goiardi.search_items; VACUUM;' | /usr/bin/psql -d <GOIARDI_DB_NAME>` to the crontab of your postgres user (probably postgres), or some other user account with rights to the goiardi database, and that will take care of the reindexing for you. Running this command daily is a good idea, but you can experiment with reindexing at different time frames and see what works best for you. The act of reindexing itself does not appear to be particularly stressful, but of course finding a relatively quiet time to do the reindexing is probably a good idea.

Also note that as this is a pretty feature the details are subject to change. In particular, the indexes on the search_items table are likely not to be optimal; you should experiment with tweaking those as you see fit, and if you find something (or the removal of something) that works especially well, please let me know.

This is very new, and while it's been tested pretty thoroughly and has been running reliably in production for a while it may still have some problems. If so, filing issues is appreciated.

10.3 Search index trimming

One option added in version 0.11.3 is the ability to trim the length of values (not keys) that will be stored in the index with `-T/--index-val-trim`. This leads to smaller indexes and, hopefully, lower memory usage. Currently, it defaults to 0 (meaning that no values in the index will be trimmed), but this behavior will change with the next major release.

Some thought should be put in to what the trim length should be. If it's too short, searches may have unexpected problems. In testing with chef-pedant locally, trimming values down to 50 characters caused some search tests to break, while 100 characters worked fine. A good value generally is 100 characters, but you may need to adjust the trim value and test until you find a good number if 100 characters doesn't work well for you.

CHAPTER 11

Event Logging

Goiardi has optional event logging. When enabled with the `--log-events` command line option, or with the `"log-events"` option in the config file, changes to clients, users, cookbooks, data bags, environments, nodes, and roles will be tracked. The event log can be viewed through the /events API endpoint.

If the `-K`/`--log-event-keep` option is set, then once a minute the event log will be automatically purged, leaving that many events in the log. This is particularly recommended when using the event log in in-memory mode.

If the `--skip-log-extended` option is set, then the JSON encoded blob of the object being logged will not be stored.

The easiest way to use the event log is with the knife-goiardi-event-log knife plugin. It's available on rubygems, or at github at https://github.com/ctdk/knife-goiardi-event-log.

The event API endpoints work as follows:

- `GET /events` - optionally taking `offset`, `limit`, `from`, `until`, `object_type`, `object_name`, and `doer` query parameters.

  List the logged events, starting with the most recent. Use the `offset` and `limit` query parameters to view smaller chunks of the event log at one time. The `from`, `until`, `object_type`, `object_name`, and `doer` query parameters can be used to narrow the results returned further, by time range (for `from` and `until`), the type of object and the name of the object (for `object_type` and `object_name`) and the name of the performer of the action (for `doer`). These options may be used in singly or in concert.

- `DELETE /events?purge=1234` - purge logged events older than the given id from the event log.
- `GET /events/1234` - get a single logged event with the given id.
- `DELETE /events/1234` - delete a single logged event from the event log.

A user or client must be an administrator account to use the `/events` endpoint.

The data returned from the event log should look something like this:

```
{
  "actor_info": "{\"username\":\"admin\",\"name\":\"admin\",\"email\":\"\",\"admin\
↪":true}\n",
  "actor_type": "user",
```

```
  "time": "2014-05-06T07:40:12Z",
  "action": "delete",
  "object_type": "*client.Client",
  "object_name": "pedant_testclient_1399361999-483981000-42305",
  "extended_info": "{\"name\":\"pedant_testclient_1399361999-483981000-42305\",\"node_
↪name\":\"pedant_testclient_1399361999-483981000-42305\",\"json_class\":\
↪"Chef::ApiClient\",\"chef_type\":\"client\",\"validator\":false,\"orgname\":\
↪"default\",\"admin\":true,\"certificate\":\"\"}\n",
  "id": 22
}
```

CHAPTER 12

Reporting

Goiardi now supports Chef's reporting facilities. Nothing needs to be enabled in goiardi to use this, but changes are required with the client. See http://docs.chef.io/reporting.html for details on how to enable reporting and how to use it.

There is a goiardi extension to reporting: a "status" query parameter may be passed in a GET request that lists reports to limit the reports returned to ones that match the status, so you can read only reports of chef runs that were successful, failed, or started but haven't completed yet. Valid values for the "status" parameter are "started", "success", and "failure".

To use reporting, you'll either need the Chef knife-reporting plugin, or use the knife-goiardi-reporting plugin that supports querying runs by status. It's available on rubygems, or on github at https://github.com/ctdk/knife-goiardi-reporting.

12.1 Purging Reports and Statuses

If you'd like to purge reports and node statuses after a period of time, the `--purge-reports-after` and `--purge-status-after` arguments are available. Given a period of time in Golang duration format (like `"720h"`), goiardi will periodically purge reports and statuses older than that time. If it's not set they will be kept forever.

CHAPTER 13

Berks Universe Endpoint

Starting with version 0.6.1, goiardi supports the berks-api `/universe` endpoint. It returns a JSON list of all the cookbooks and their versions that have been uploaded to the server, along with the URL and dependencies of each version. The requester will need to be properly authenticated with the server to use the universe endpoint.

The universe endpoint works with all backends, but with a ridiculous number of cookbooks (like, loading all 6000+ cookbooks in the Chef Supermarket), the Postgres implementation is able to take advantage of some Postgres specific functionality to generate that page significantly faster than the in-mem or MySQL implementations. It's not too bad, but on my laptop at home goiardi could generate /universe against the full 6000+ cookbooks of the supermarket in ~350 milliseconds, while MySQL took about 1 second and in-mem took about 1.2 seconds. Normal functionality is OK, but if you have that many cookbooks and expect to use the universe endpoint often you may wish to consider using Postgres.

CHAPTER 14

Serf

As of version 0.8.0, goiardi has some serf integration. At the moment it's mainly used for shovey (see below), but it will also announce that it's started up and joined a serf cluster.

If the `--serf-event-announce` flag is set, goiardi will announce logged events from the event log and starting up and joining the serf cluster over serf as serf user events. Be aware that if this is enabled, something will need to read these events from serf. Otherwise, the logged events will pile up and eventually take up all the space in the event queue and prevent any new events from being added.

CHAPTER 15

Shovey

Shovey is a facility for sending jobs to nodes independently of a chef-client run, like Chef Push but serf based.

15.1 Shovey requirements

To use shovey, you will need:

- Serf installed on the server goiardi is running on.
- Serf installed on the node(s) running jobs.
- `schob`, the shovey client, must be installed on the node(s) running jobs.
- The `knife-shove` plugin must be installed on the workstation used to manage shovey jobs.

The client can be found at https://github.com/ctdk/schob, and a cookbook for installing the shovey client on a node is at https://github.com/ctdk/shovey-jobs. The `knife-shove` plugin can be found at https://github.com/ctdk/knife-shove or on rubygems.

15.2 Shovey Installation

Setting goiardi up to use shovey is pretty straightforward.

- Once goiardi is installed or updated, install serf and run it with `serf agent`. Make sure that the serf agent is using the same name for its node name that goiardi is using for its server name.
- Generate an RSA public/private keypair. Goiardi will use this to sign its requests to the client, and schob will verify the requests with it.:

```
openssl genrsa -out shovey.pem 2048 # generate 2048 bit private key
openssl rsa -in shovey.pem -pubout -out shovey.key # public key
```

  Obviously, save these keys.

- If you're using an external service (like vault) to store secrets, please see *Secret Handling* for how to set up shovey's signing key with that.
- Run goiardi like you usually would, but add these options: `--use-serf --use-shovey --sign-priv-key=/path/to/shovey.pem`
- Install serf and schob on a chef node. Ensure that the serf agent on the node is using the same name as the chef node. The `shovey-jobs` cookbook makes installing schob easier, but it's not too hard to do by hand by running `go get github.com/ctdk/schob` and `go install github.com/ctdk/schob`.
- If you didn't use the shovey-jobs cookbook, make sure that the public key you generated earlier is uploaded to the node somewhere.
- Shovey uses a whitelist to allow jobs to run on nodes. The shovey whitelist is a simple JSON hash, with job names as the keys and the commands to run as the values. There's a sample whitelist file in the schob repo at `test/whitelist.json`, and the shovey-jobs cookbook will create a whitelist file from Chef node attributes using the usual precedence rules. The whitelist is drawn from `node["schob"]["whitelist"]`.
- If you used the shovey-jobs cookbook schob should be running already. If not, start it with something like `schob -VVVV -e http://chef-server.local:4545 -n node-name.local -k /path/to/node.key -w /path/to/schob/test/whitelist.json -p /path/to/public.key --serf-addr=127.0.0.1:7373`. Within a minute, goiardi should be aware that the node is up and ready to accept jobs.

At this point you should be able to submit jobs and have them run. The knife-shove documentation goes into detail on what actions you can take with shovey, but to start try `knife goiardi job start ls <node name>`. To list jobs, run `knife goiardi job list`. You can also get information on a shovey job, detailed information of a shovey job's run on one node, cancel jobs, query node status, and stream job output from a node with the knife-shove plugin. See the plugin's documentation for more information.

See the serf docs at http://www.serfdom.io/docs/index.html for more information on setting up serf. One serf option you may want to use, once you're satisfied that shovey is working properly, is to use encryption with your serf cluster.

15.3 Shovey In More Detail

Every thirty seconds, schob sends a heartbeat back to goiardi over serf to let goiardi know that the node is up. Once a minute, goiardi pulls up a list of nodes that it hasn't seen in the last 10 minutes and marks them as being down. If a node that is down comes back up and sends a heartbeat back to goiardi, it is marked as being up again. The node statuses are tracked over time as well, so a motivated user could track node availability over time.

When a shovey run is submitted, goiardi determines which nodes are to be included in the run, either via the search function or from being listed on the command line. It then sees how many of the nodes are believed to be up, and compares that number with the job's quorum. If there aren't enough nodes up to satisfy the quorum, the job fails.

If the quorum is satisfied, goiardi sends out a serf query with the job's parameters to the nodes that will run the shovey job, signed with the shovey private key. The nodes verify the job's signature and compare the job's command to the whitelist, and if it checks out begin running the job.

As the job runs, schob will stream the command's output back to goiardi. This output can in turn be streamed to the workstation managing the shovey jobs, or viewed at a later time. Meanwhile, schob also watches for the job to complete, receiving a cancellation command from goiardi, or to timeout because it was running too long. Once the job finishes or is cancelled or killed, schob sends a report back to goiardi detailing the job's run on that node.

CHAPTER 16

The Shovey API

Documentation of the goiardi shovey HTTP and Serf APIs. As a work in progress, be aware that anything in this document is subject to change until shovey is officially released. This documentation covers all the endpoints shovey uses, both on the goiardi side and the schob (the shovey client that executes jobs) side, but at the moment is a little sparse. This will change as the documentation fills out.

16.1 HTTP API

The Chef Pushy API located at http://docs.getchef.com/push_jobs.html#api-push-jobs is also relevant, but the Shovey HTTP API is not exactly the same as the Pushy API, for various reasons.

16.2 Shovey job control

`/shovey/jobs`

Methods: GET, PUT

- Method: GET

  List all jobs on the server. Returns a list of uuids of jobs.

  Response body format:

```
[
  "036d8b61-da10-439b-ba1f-40f5f866c6b1",
  "04226cc8-0c9b-47e5-adaa-158ccc36f0b1",
  "1204692a-8e4c-4adb-960a-089d59c10fbf",
  "242957ce-10c5-4f7a-89d8-ffb478fd1ef9"
]
```

- Method: POST

  Create a new shovey job.

Request body format:

```
{
    "command": "foo",
    "quorum": "75%",
    "nodes": [ "foo.local", "bar.local" ]
}
```

Response body format:

```
{
    "id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
    "uri": "http://your.chef-server.local:4545/shovey/jobs/76b745eb-45d6-4856-
↪94f9-7830e79cb8cd"
}
```

`/shovey/jobs/<JOB ID>`

- Method: GET

  Information about a shovey job's status, both overall and each node's status.

  Response body format:

```
{
  "command": "ls",
  "created_at": "2014-08-26T21:44:24.636242093-07:00",
  "id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
  "nodes": {
    "succeeded": [
      "nineveh.local"
    ]
  },
  "run_timeout": 300,
  "status": "complete",
  "updated_at": "2014-08-26T21:44:25.079010129-07:00"
}
```

`/shovey/jobs/<JOB ID>/<NODENAME>`

Methods: GET, PUT

- Method: GET

  Provides detailed information about a shovey run on a specific node.

  Response body format:

```
{
  "run_id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
  "node_name": "nineveh.local",
  "status": "succeeded",
  "ack_time": "2014-08-26T21:44:24.645047317-07:00",
  "end_time": "2014-08-26T21:44:25.078800724-07:00",
  "output": "Applications\nLibrary\nNetwork\nSystem\nUser␣
↪Information\nUsers\nVolumes\nbin\ncores\ndev\netc\nhome\nmach_
↪kernel\nnet\nopt\nprivate\nsbin\ntmp\nusr\nvar\n",
  "error": "",
  "stderr": "",
  "exit_status": 0
}
```

- Method: PUT

  Update a node's shovey run information on the server.

  Request body format:

```
{
  "run_id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
  "node_name": "nineveh.local",
  "status": "succeeded",
  "ack_time": "2014-08-26T21:44:24.645047317-07:00",
  "end_time": "2014-08-26T21:44:25.078800724-07:00",
  "error": "",
  "exit_status": 0,
  "protocol_major": 0,
  "protocol_minor": 1
}
```

  Response body format:

```
{
  "id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
  "node": "nineveh.local",
  "response": "ok"
}
```

`/shovey/jobs/cancel`

Methods: PUT

- Method: PUT

  Cancels a job. The "nodes" option can either be a list of nodes to cancel the job on, or use an empty array to cancel the job on all nodes running this job.

  Request body format:

```
{
  "run_id": "76b745eb-45d6-4856-94f9-7830e79cb8cd",
  "nodes": [ "foomer.local", "noober.snerber.com" ]
}
```

  Response body format:

```
{
  "command"=>"sleepy",
  "created_at"=>"2014-08-26T21:55:07.751851335-07:00",
  "id"=>"188d457e-2e07-40ef-954c-ab936af615b6",
  "nodes"=>{"cancelled"=>["nineveh.local"]},
  "run_timeout"=>300,
  "status"=>"cancelled",
  "updated_at"=>"2014-08-26T21:55:25.161713014-07:00"
}
```

16.3 Streaming output

`/shovey/stream/<JOB ID>/<NODE>`

Methods: GET, PUT

- Method: GET

  Streams the output from a job running on a node. Takes two query parameters: `sequence` and `output_type`. The `sequence` parameter is the the sequence record to start fetching from, while `output_type` sets the sort of output you'd like to receive. Acceptable values are 'stdout', 'stderr', and 'both'. The default value for `sequence` if none is given is 0, while the default for `output_type` is 'stdout'.

  Response body format:

```
{
  "run_id": "188d457e-2e07-40ef-954c-ab936af615b6",
  "node_name": "foomer.local",
  "last_seq": 123,
  "is_last": false,
  "output_type": "stdout",
  "output": "foo"
}
```

- Method: PUT

  Add a chunk of output from a shovey job on a node to the log on the server for the job and node.

  Request body format:

```
{
  "run_id": "188d457e-2e07-40ef-954c-ab936af615b6",
  "node_name": "foomer.local",
  "seq": 1,
  "is_last": false,
  "output_type": "stdout",
  "output": "foo"
}
```

  Response body format:

```
{
  "response":"ok"
}
```

16.4 Node status

`/status/all/nodes`

Methods: GET

- Method: GET

  Get the latest status from every node on the server.

  Response Body format:

```
[
  {
    "node_name": "nineveh.local",
    "status": "up",
    "updated_at": "2014-08-26T21:49:58-07:00",
    "url": "http://nineveh.local:4545/status/node/nineveh.local/latest"
  },
  {
```

```
    "node_name": "fooper.local",
    "status": "down",
    "updated_at": "2014-08-26T21:47:48-07:00",
    "url": "http://nineveh.local:4545/status/node/fooper.local/latest"
  }
]
```

`/status/node/<NODENAME>/all`

Methods: GET

- Method: GET

  Get a list of all statuses a particular node has had.

  Response body format:

```
[
  {
    "node_name": "nineveh.local",
    "status": "up",
    "updated_at": "2014-08-26T21:51:28-07:00"
  },
  {
    "node_name": "nineveh.local",
    "status": "up",
    "updated_at": "2014-08-26T21:50:58-07:00"
  },
  {
    "node_name": "nineveh.local",
    "status": "up",
    "updated_at": "2014-08-26T21:50:28-07:00"
  },
  {
    "node_name": "nineveh.local",
    "status": "up",
    "updated_at": "2014-08-26T21:49:58-07:00"
  }
]
```

`/status/node/<NODENAME>/latest`

Methods: GET

- Method: GET

  Get the latest status of this particular node.

  Response body format:

```
{
  "node_name": "nineveh.local",
  "status": "up",
  "updated_at": "2014-08-26T21:50:58-07:00"
}
```

CHAPTER 17

serf API

17.1 Node status

Sent by schob to goiardi over serf as a heartbeat message.

Serf parameters:

- Name: node_status
- Payload: JSON described below
- RespCh: goiardi will respond to the heartbeat message over this response channel.

JSON payload parameters:

- node: name of the chef client/node.
- status: "up"

17.2 Shovey command

Sent by goiardi to schob over serf to start a shovey run on a node.

Serf parameters:

- Name: "shovey"
- Payload: JSON described below
- FilterNodes: Limit the serf query to the given nodes
- RequestAck: request an acknowledgement from schob
- AckCh, RespCh: acknowledgement and response channels from schob to goiardi.

JSON payload parameters:

- run_id: the uuid of the shovey run

- action: the action to perform on the node. May be “start” or “cancel”.
- command: the name of the command to run. Only required when action is “start”.
- time: RFC3339 formatted current timestamp
- timeout: Time, in seconds, to kill the process if it hasn’t finished by the time the timeout expires.
- signature: assembled from the JSON payload by joining the elements of the JSON payload that aren’t the signature, separated by newlines, in alphabetical order. The goiardi server must be given an RSA private key to sign the request with, and schob must have the public key matching that private key to verify the request.

The block to sign will look something like this:

- action: start
- command: foo
- run_id: b5a6ee64-67ca-4a4f-94ad-6c18eb1c6a32
- time: 2014-09-05T23:00:00Z
- timeout: 300

CHAPTER 18

Logging

By default, goiardi logs to standard output. A log file may be specified with the `-L/--log-file` flag, or goiardi can log to syslog with the `-s/--syslog` flag on platforms that support syslog. Attempting to use syslog on a platform that doesn't support syslog (currently Windows and plan9 (although plan9 doesn't build for other reasons)) will result in an error.

18.1 Log levels

Log levels can be set in goiardi with either the `log-level` option in the configuration file, the `--log-level` flag on the command line, the `$GOIARDI_LOG_LEVEL` environment variable, or with one to five `-V` flags on the command line. Log level options are "debug", "info", "warning", "error", "critical", and "fatal". More `-V` on the command line means more spewing into the log.

CHAPTER 19

WebUI

Until such time that goiardi finally reaches 1.0.0 and can use the official Chef management console (free for up to 25 nodes), goiardi can use the old Chef WebUI if you're so inclined. Goiardi can use webui without any tweaks, but there's a forked webui repo at https://github.com/ctdk/chef-server-webui with some customizations to make it a little bit easier to use. There's not currently a smooth easy way to install this webui yet, unfortunately, but it's just a basic Rails app. It hasn't been merged into master yet, but there's a webui installation recipe you can look at at https://github.com/ctdk/chef-goiardi/tree/rt-work for guidance until a smoother procedure is worked out.

Chef-browser is another web based frontend for Chef, available at https://github.com/3ofcoins/chef-browser, that works with goiardi. It doesn't have all the features of chef-webui, but it's very nice on its own merits.

CHAPTER 20

Metrics

Starting with goiardi v0.10.4, goiardi can export metrics about itself via statsd. In turn, statsd can feed these metrics into a time series database like graphite. Once in graphite, one could visualize the data with something like grafana, or set up alerts with that data in bosun.

At this time, goiardi exports via statsd metrics covering the runtime (memory usage, garbage collection, goroutines), API timing, information about chef-client runs, the number of nodes, and search timing.

The available metrics via statsd currently are:

- `node.count` - number of nodes currently in the system
- `runtime.goroutines` - number of goroutines running
- `runtime.memory.allocated` - allocated memory in bytes
- `runtime.memory.mallocs` - number of mallocs
- `runtime.memory.frees` - number of times memory's been freed
- `runtime.memory.heap` - size of heap memory in bytes
- `runtime.memory.stack` - size of stack memory in bytes
- `runtime.gc.total_pause` - how many nanoseconds goiardi has paused for garbage collection the whole time the process has been running.
- `runtime.gc.pause_per_sec` - pauses per second
- `runtime.gc.pause_per_tick` - pauses per interval sending metrics to statsd (currently 10 seconds)
- `runtime.gc.num_gc` - number of garbage collections
- `runtime.gc.gc_per_sec` - gc per second
- `runtime.gc.gc_per_tick` - gc per statsd tick (as above, every 10 secodns)
- `runtime.gc.pause` - timing of how long each gc pause lasts
- `api.timing.%s.%s`, where "`%s.%s`" is the first part of the api endpoint path and the HTTP method (so, for example, a PUT to cookbooks would be `api.timing.cookbooks.put`) - timing of API endpoint requests

- `client.run.started` - Count of started chef-client runs
- `client.run.success` - Count of successful chef-client runs
- `client.run.failure` - Count of failed chef-client runs
- `client.run.run_time` - Timing of how long
- `client.run.total_resource_count` - Total resources in a run
- `client.run.updated_resources` - Total updated resources in a run
- `search.in_mem` - timing of in-memory searches
- `search.pg` - timing of Postgres-based searches

CHAPTER 21

S3 File Uploads

With goiardi 0.11.0, you can store cookbook files in S3 (or a compatible service), instead of storing them locally.

21.1 Configuration

21.1.1 goiardi options

There are five options to set for goiardi S3 cookbook uploads. They are:

- `use-s3-upload`: Enables (or disables) the S3 cookbook uploads.
- `aws-region`: The AWS region to upload files to. No default.
- `s3-endpoint`: An optional setting to change the S3 endpoint. Defaults to `s3.amazonaws.com`, which is what you want, but can be set to any S3-compatible service.
- `aws-disable-ssl`: Disable SSL with S3 URLs. Almost certainly not something you should enable, unless you're running fakes3 or somesuch locally for testing.
- `s3-file-period`: How long, in minutes, links to cookbook files should remain valid to upload/download. Defaults to 15.

These options can be set on the command line or in the goiardi.conf file.

21.1.2 S3 credentials

There are a few ways of storing the AWS credentials for goiardi to use for the S3 uploads.

The best way is to use the AWS credentials file.

Failing that, set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables. They can be added to the init script that starts goiardi, or it can be added to the goiardi config file in the `env-vars` section.

21.2 Testing it out

Fake-s3 works pretty well for testing, but the current master doesn't handle bulk deletes from the aws-sdk go client. To fix this, apply the patch below to fake-s3, rebuild the gem, and reinstall.

```
diff --git a/lib/fakes3/server.rb b/lib/fakes3/server.rb
index 47a4456..e4f3119 100644
--- a/lib/fakes3/server.rb
+++ b/lib/fakes3/server.rb
@@ -257,6 +257,16 @@ module FakeS3
         )

         response.body = XmlAdapter.complete_multipart_result real_obj
+      elsif query.has_key?('delete')
+       keys = s_req.webrick_request.body.scan(/\<Key\>(.*?)\<\/Key\>/).flatten
+
+       bucket_obj = @store.get_bucket(s_req.bucket)
+       keys.each do |k|
+         @store.delete_object(bucket_obj, k, s_req.webrick_request)
+       end
+
+       response.status = 204
+       response.body = ""
       elsif request.content_type =~ /^multipart\/form-data; boundary=(.+)/
         key=request.query['key']
```

If you're using fake-s3 to test it out, set the `--aws-disable-ssl` option (if you don't run fake-s3 with SSL enabled) and set `--s3-endpoint` to the bucket + fake-s3's hostname. (If you're using `fakes3.local` for your fake-s3 server and you're using a bucket named `goiardi`, add `goiardi.fakes3.local` to `/etc/hosts`.)

21.3 Converting from local file store to S3

A script is provided in `scripts/lsf-conv.sh` to make the local filestore to S3 transition easier. It requires s3cmd, but could easily be adapted to use the official aws-cli.

Once your bucket and s3cmd are all configured, run the script to upload the cookbook files. The options are:

`Usage: -a <AWS access id> -s <AWS secret> -r <region> -b <bucket> -d <local filestore directory>`

The AWS access id and secret are optional if you're using the same credentials for chef as you've configured s3cmd to use.

Nothing's in place for converting back to the local filestore from S3, but it wouldn't be too hard. All you would need to do is download all of the files from S3 and make sure they all get back into the local filestore directory (rather than the subdirectories derived from the first two letters of the hash).

CHAPTER 22

Secret Handling

Starting with version 0.11.1, goiardi can use external services to store secrets like public keys, the signing key for shovey, and user password hashes. As of this writing, only Hashicorp's vault is supported. This is very new functionality, so be aware.

NB: If goiardi has been compiled with the `novault` build tag, none of this will be available.

22.1 Configuration

The relevant options for secret configuration on goiardi's end are:

- `--use-external-secrets`: Turns on using an external secret store.
- `--vault-addr=<address>`: Address of vault server. Defaults to the value the `VAULT_ADDR` environment variable, but can be specified here. Optional.
- `--vault-shovey-key=<path>`: Optional path for where shovey's signing key will be stored in vault. Defaults to "keys/shovey/signing". Only meaningful, unsurprisingly, if shovey is enabled.

Each of the above command-line flags may also be set in the configuration file, with the `--` removed.

Additionally, the `VAULT_TOKEN` environment variable needs to be set. This can either be set in the configuration file in the `env-vars` stanza in the configuration file, or exported to goiardi in one of the many other ways that's possible.

To set up vault itself, see the intro and the general documentation for that program. For goiardi to work right with vault, there will need to be a backend mounted with `-path=keys` before goiardi is started.

22.2 Populating

A new goiardi installation won't need to do anything special to use vault for secrets - assuming everything's set up properly, new clients and users will work as expected.

Existing goiardi installations will need to transfer their various secrets into vault. A persistent but not DB backed goiardi installation will need to export and import all of goiardi's data. With MySQL or Postgres, it's much simpler.

For each secret, get the key or password hash from the database for each object and make a JSON file like this:

```
{
        "secretType": "secret-data\nwith\nescaped\nnew\nlines\nif-any"
}
```

(Once everything looks good with the secrets being stored in vault, those columns in the database should be cleared.)

The "secretType" is "pubKey" for public keys, "passwd" for password hashes, and "RSAKey" for the shovey signing key.

Optionally, you can add a `ttl` (with values like "60s", "30m", etc) field to that JSON, so that goiardi will refetch the secret after that much time has passed.

Now this JSON needs to be written to the vault. For client and user public keys, the path is "keys/clients/<name>" for clients and "keys/users/<name>" for users. User password hashes are "keys/passwd/users/<name>". The shovey signing key is more flexible, but defaults to "keys/shovey/signing". If you save the shovey key to some other path, set `--vault-shovey-key` appropriately.

CHAPTER 23

CHANGELOG

The goiardi CHANGELOG for all of its various releases.

```
0.11.7
------
* Allow access to /debug/pprof with a whitelist of IP addresses
* Properly index arrays of hashes, arrays of arrays, etc. in object attributes.
* Pretty serious memory usage improvements with search (both the in-memory
  and postgres searches).
* Fix reconnecting to serf if the connection is somehow interrupted.
* Fix negated range queries (it turns out they *do* have a use after all), and
  refactor how NOT queries are handled generally.
* Add options to purge old reports and node statuses.
* Add option to skip logging extended object information in the event log.
* A handful of other bugfixes.
* Bump up to using golang 1.9.3 for builds.
* Minor changes to the documentation.

0.11.6 (cancelled)
------
* Skipped because of a miscommunication snafu involving Debian packaging and a
  pre-release tag for 0.11.6.

0.11.5
------
* Several search fixes:
  - With postgres search:
    * Fixed reindexing after it broke with the previous update that eliminated
    a lot of unneeded extra rows in the database.
    * Fixed basic queries with NOT statements.
    * Separately, fixed using NOT with subqueries. On a somewhat complicated
      note, but in a way that appears to match standard Solr behavior, when
      doing a query like "name:chef* AND NOT (admin:true OR admin:bleh)" it
      works as is, but when a negated subquery is followed by another basic
      query statement, it needs to have extra parentheses around the NOT +
      subquery, like "name:chef* AND (NOT (admin:true OR admin:bleh)) AND
```

```
      public_key:*". A convoluted and unlikely scenario, but it could happen.
  - With in-memory search:
    * NOT + subqueries was also broken with the in-mem search. The fixes for
      the pg-search partially fixed it for in-mem in that it no longer made the
      server panic, but it was returning incorrect results. Additional work
      ended up being needed for in-mem search.

0.11.4
------
* Implement Chef authentication version 1.3.
* Move the custom goiardi error type out of util and into its own module.
  Wrappers around the new module are in util still for convenience, and
  because the functions and interface are used all over the place.
* Many endpoints now handle HEAD requests where appropriate. With some
  endpoints this is not especially useful, but with others it's a lightweight
  way to see what resources exist and so forth. Implements Chef RFC 090.
* Start using contexts with requests. This does mean that goiardi will require
  at least go 1.7. (As of 0.11.3 goiardi only supported go 1.7+, but it was
  likely to build with somewhat older versions anyway.)
* Minor bugfixes - deal with a possible race condition with the in-mem search
  index, change some logging statements from Info to Debug that didn't need to
  be Info level and removed a test log statement that was no longer necessary,
  updated copyright dates.
* Add the Chef API version header to responses.
* Change behavior if the data file and use-(mysql|postgresql) are specified
  together; formerly it was a fatal error, but now it'll just emit a warning
  in the error log and ignore the data file setting.

0.11.3
------
* Add an option to trim values in search indexes. Currently not enabled by
  default, but will be in the next minor goiardi release (so, either 0.12.0 or
  1.0.0, depending on which ends up being next). Existing indexes ought to be
  reindexed upon upgrading, but they should still work if this is skipped.
* Fix a bug where duplicated items in slices in objects being indexed with the
  in-memory trie based index would cause goiardi to crash. For good measure,
  even though it isn't necessary to prevent a crash remove those same
  duplicate items from objects being indexed with the postgres index.
* Mark --use-unsafe-mem-store as deprecated. In the unlikely event someone's
  using that option, a warning will print in the log. This option may be
  removed at any time.
* Allow setting configuration options via environment variables. (See
  the documentation for the details.)
* Finally allow configuring MySQL or PostgreSQL connection options with
  command line flags (or, now, environment variables).
* Fixed format issues and wording in a few places in the documentation, along
  with updating the docs for the current version.
* Add a hidden flag to generate a simple man page.
* Add that simple man page, along with the html docs, to the packagecloud.io
  packages.
* Add a Dockerfile to allow running the local goiardi source in docker.
* Add Debian "stretch" and Ubuntu "yakkety yak" to the distro versions we have
  in the package repository.

0.11.2
------
* Fix a bug with escaped characters in certain searches (thanks ickymettle).
  Does require rebuilding the search index.
```

```
* Allow using 'novault' as a build tag to avoid having to have the vault api
  present when building goiardi. Not relevant to most people.


0.11.1
------
* Allow storing secrets (client & user public keys, shovey signing private
  keys, and user password hashes) in an external service. Currently only vault
  is supported.
* Rework reindexing to break it into smaller chunks and ensure that only one
  reindexing job can run at a time.
* Package goiardi for RHEL 7 and Debian jessie for s390x. Rather experimental,
  of course.

0.11.0
------
* Ability to upload cookbooks to S3.
* Add script to upload local files to S3 to migrate.
* Change how items are indexed with the postgres indexer, to reduce the number
  of rows in the search_items table substantially (at the cost of possible
  differences in search results in a few weird corner cases).
* Search parser no longer chokes on Unicode. Unfortunately Postgres' ltree
  module does not accept all Unicode alphanumeric characters as valid still.
* Use vendoring.
* Rejigger the package building process a bit - changing how the different
  packages are built and how version numbers are determined.
* Fix a long-standing annoyance where the log file would get truncated when
  goiardi started or restarted.
* Allow passing environment variables to goiardi through the config file.
* Fix in-memory indexer to work with go 1.7.
* Add packages for CentOS 6 and 7. Also use a gox fork pulling in someone's PR
  with better ARM support until that gets merged upstream eventually.
* Change the postgres columns using the 'json' data type to use 'jsonb'
  instead. This is generally better, but does mean that goiardi now requires
  PostgreSQL 9.4 or later.

0.10.4
------
* Export pprof info over HTTP, but only accept connections from localhost for
  that information.
* Add statsd metrics for things like chef-client run timings (requires
  reporting) and started/succeeded/failed, number of nodes, API endpoint
  timings, various pieces of runtime info like GC pauses, RAM used, and number
  of resources updated & total resources for client runs.
* Fix JSON decoding issue where very large numbers would suddenly turn into
  floats.

0.10.3
------
* Handle someone trying to use syslog on Windows ourselves, rather than
  letting the logging library do it (it was causing trouble with gox).

0.10.2
------
* Fix up packaging and deploy scripts a bit
* Add sql schemas to the deb
* Fixed a logic error when configuring the address to listen on where the
  value specified in the config file was always ignored, and only an address
```

```
  specified on the command line worked. (Thanks to jordi and DQEbert here for
  bringing this to my attention.)
* Added options to specify proxy hostname and port different than what goairdi
  itself is listening on. (Thanks to jordi and DQEbert here as well.)
* CoC
* Added Debian wheezy to the list of distros we generate packages for.
* The logging library goiardi used moved. It had been forked, but since the
  dependencies of said fork also moved, goiardi switched to the new version of
  that library. Happily the logger library had added logging to syslog as an
  option, so we just went back to using upstream at the new location.  (Thanks
  to theckman for providing a fix for this.)
* In concert with the above, add a "fatal" log level.
* Terraform removed the depgraph module, so that's been vendored into goiardi
  along with its digraph dependency.

0.10.1
------
* Fix some tests
* Scripts, configuration files for more efficient packaging
* circleci integration
* Bomb on importing data if public keys don't validate. (thanks jordi and
  DQEbert for bringing this to my attention.)
* Validate older PKCS#1 keys -- golang's stdlib pukes on them without some
  massaging. (thanks jordi and DQEbert for bringing this to my attention.)
* Fix reindexing - databags were not being reindexed with the postgres search,
  and the SaveItem calls were moved to goroutines; otherwise, the request
  from knife would time out and knife would restart the reindex.
* Allow '.' in cookbook names; despite what an error chef-pedant is looking
  for, those are allowed. (thanks jordi and DQEbert for bringing this to my
  attention.)
* Make the authentication lib more general (thanks theckman)
* Output the version of golang used to build a particular goiardi binary
  (again, thanks theckman)
* The changed hostname in URLs to download bug didn't get fixed in 0.7.1 quite
  all the way after all. It is now. (Thanks to oker1 for bringing that to my
  attention.)
* Fixed search tests to pass when run using more than one processor. (Brought
  to my attention by theckman.)
* Fixed a deadlock that could happen when saving an in-mem index to disk at
  the exact moment an object was being indexed. Seems to be specific to
  go1.5.1 (or at least it never happened before that I saw), but needs fixed
  anyway. (Also brought to my attention by theckman.)
* Fixed broken pipe errors with too large requests when running chef-pedant
  against goiardi built with go 1.5.1.
* Update some docs.

0.10.0
------
* Search architecture changed so different search backends can be used (thanks
  oker1 for your work on that).
* Postgres search is here at last! If you're using Postgres, instead of using
  the ersatz solr search, you can instead use Postgres to power your searches.
* Add a mutex for the original goiardi search - multiple simple queries
  executing simultaneously are not a problem, but multiple complex queries can
  eat up all the RAM on the machine and cause goiardi to crash. This mitigates
  that situation.
* Be a little more forgiving with reporting protocol versions - allow
  specifying the protocol version as a query param instead of only as a
```

```
  header. This is to make showing reports with the webui a little easier.
* Bump the Chef Server version we claim to be from 11.1.6 to 11.1.7.

0.9.2
-----
* Fix broken import/export function with reports - bringing goiardi's variable
  naming inline with golang conventions a while back inadvertently renamed a
  reporting JSON field. The field was renamed, and the import code will now
  handle both correct and incorrect names for the node reporting.

0.9.1
-----
* Fix error where requests for zero byte cookbook files would crash.
* Authentication docs improvments (thanks oker1!)
* Rewritten and more robust cookbook depsolver.
* Fix for client creation with cheffish (thanks whiteley!)
* Fix for search where searching for something like "foo:bar AND NOT foo:bar"
  was returning incorrect results. (brought to my attention and test provided
  by brimstone, thanks!)
* Fixed a bug where clients could be created with the same name as a user (or
  vice versa) in in-memory mode.

0.9.0
-----
* Validate IP address supplied on the command line or in the config file.
* Compress index docs to reduce memory usage with the search index.
* Ordering searches works now.
* Index and datastore files now only write to disk if there have been changes
  since the last time they were saved.
* In tandem with the previous change, freeze interval default has been changed
  from 300 seconds to 10 seconds.
* Bump Chef Server version we claim to be from 11.1.3 to 11.1.6.

0.8.2
-----
* Fix typo with checking for an existing client in SQL mode.
* Fix typo in sample config file for postgres option.
* Add additional checks to the local filestore option to make sure the supplied
  directory name exists and is a directory.

0.8.1
-----
* Disable SSLv3 when using TLS.
* The main goiardi docs are now located at http://goiardi.readthedocs.org/en/latest/

0.8.0
-----
* Introducing shovey, a facility for running commands on nodes without a full
  chef run.
* Goiardi can act as a serf client now. Mostly this is for shovey support, but
  it can also optionally announce logged events and startup over serf as serf
  events.
* If serf is used, node statuses will be tracked by goiardi. This depends on
  receiving a heartbeat message from the shovey client.
* Add an error for the unlikely situation where an SQL function is called, yet
  no SQL database is configured.

0.7.2
```

```
-----
* Remove a newline in a debug statement, courtesy of @spheromak.
* Also per @spheromak's suggestion, fixed some possible race conditions
  revealed by building goiardi with the -race flag and running chef-pedant
  against it.
* Edit doc.go slightly to make godocs more attractive.

0.7.1
-----
* Add --db-pool-size and --max-connections options for configuring the number
  of idle db connections kept around and the maximum number of db connections
  to make to the server. It isn't particularly useful if you're not using one
  of the SQL backends.
* For locally stored cookbook files (which is currently all of them), goiardi
  now generates the URL to the resource from the currently configured
  hostname. This fixes an issue where if you uploaded a cookbook and then
  changed the goairdi server's hostname, the URLs to download cookbooks would
  break.

0.7.0
-----
* Add /universe API endpoint, per
  https://github.com/opscode/chef-rfc/blob/master/rfc014-universe-endpoint.md.
* Make file uploading a little more forgiving.
* Make validating some cookbook metadata more forgiving, to bring goiardi's
  validations in line with erchef.
* Added some functions to make listing all cookbooks and recipes on the
  server faster and move the logic into the cookbook package.
* Breaking DB change: with both MySQL and Postgres, the way data structures
  for cookbooks, nodes, etc. has changed from gob encoding to using JSON. This
  obviously breaks existing items in the database, so the following steps must
  be followed by users using either SQL backend for data storage:

    * Export their goiardi server's data with the `-x` flag.
    * Either revert all changes to the db with sqitch, then redeploy, or drop
      the database manually and recreate it from either the sqitch patches or
      the full table dump of the release (provided starting with 0.7.0)
    * Reload the goiardi data with the `-m` flag.
  See the README or the godocs for more information.

0.6.1 (cancelled)
-----
* See notes for 0.7.0

0.6.0
-----
* Postgres support.
* Fix rebuilding indexes with an SQL backend.
* Fix a bug where in MySQL mode events were being logged twice.
* Fix an annoying chef-pedant error with data bags.
* Event logging methods that are not allowed now return Method Not Allowed
  rather than Bad Request.
* Switch the logger to a fork that can be built and used with Windows that
  excludes syslog when building on Windows.
* Add basic syslog support.
* Authentication protocol version 1.2 now supported.
* Add a 'status' param to reporting, so a list of reports return by 'knife
  runs' can be narrowed by the status of the chef run (started, success, and
```

```
  failure).
* Fix an action at a distance problem with in-memory mode objects. If this
  behavior is still desirable (it seems to be slightly faster than the new way),
  it can be turned back on with the --use-unsafe-mem-store flag. This change
  DEFINITELY breaks in-mem data file compatibility. If upgrading, export your
  data, upgrade goiardi, and reload your data.
* Add several new searchable parameters for logged events.
* Add organization_id to all MySQL tables that might need it someday. Orgs are
  not used at all, so only the default value of 1 currently makes it to the
  database.
* Finally ran 'go fmt' on goiardi. It didn't even mess up the long comment
  blocks, which was what I was afraid it would do. I also ran golint against
  goiardi and took its recommendations where it made sense, which was most
  areas except for some involving generated parser code, comments on
  GobEncode/Decode, commenting a bunch of identical functions on an interface
  in search, and a couple of cases involving make and slices. All in all,
  though, the reformatting, linting, and light refactoring has done it good.

0.5.2
-----
* Add import/export of goiardi data through a JSON dump.
* Add configuration options to specify the max sizes for objects uploaded to
  the filestore and for JSON requests from the client.

0.5.1
-----
* Add log levels (from debug to critical). This makes -V/--verbose useful.
* Add an easier option in the config file to specify log levels by name.
* ipv6 already worked, but accidentally. Now it works in a more deliberate
  fashion, preventing mishaps with addresses, colons, and port numbers.
* Authentication protocol version 1.1 now supported.
* Remove a sort on run lists that was there for some reason. I have no idea
  what it was put there for, but it was wrong.
* Add an event log to log changes to objects like nodes, clients, etc. See the
  README or godocs for details.
* Add support for reporting (http://docs.opscode.com/reporting.html)

0.5.0
-----
* MySQL support added
* No longer redirect /environments/NAME/roles/NAME to
  /roles/NAME/environments/NAME
* Update documentation, reformat godocs
* Split actors apart into separate user and client types, made new Actor
  interface that encompasses both users and clients.

0.4.2
-----
Bugfix release:
* Perm tweak for nodes updating themselves.
* Small change with validating role descriptions when creating or updating
  from JSON.
* Fix issue with saving complicated indexed objects to disk where improperly
  flattened indexable objects were making the gob encoder puke all over itself
  when encoding the tries in the index docs.
* Fixed a possible regression with synchronizing cookbooks that did not show
  up in testing, but only in real use.
* An absolutely bonkers fix for listing cookbook files with webui. Webui wants
```

```
  all of the cookbook top level attributes sent over with a request to
  /cookbooks/<name>/<version>, but this is the exact *opposite* of the
  behavior chef-pedant wants, where empty definitions, attributes, etc. are
  not sent over. Knife also seems quite content with this, so the fix for now,
  since the two cases are mutually exclusionary, is to only send the empty
  hashes for those top level attributes with a GET if the request is coming
  from the webui. Bizarre, but it seems to be what's necessary.

0.4.1
-----
* Small documentation tweaks
* Fix bug with parsing config file options and rearrange setting some of those
  config struct items, fix typo in sample config file.
* Add disable-webui option for command line and config file to disable the
  chef webui rails app from connecting to goiardi.

0.4.0
-----
* Fix bug with pessimistic matching (https://github.com/ctdk/goiardi/issues/1)
* Add authentication, authorization as an option.
* Add SSL as an option.
* Fixed a few small bugs that turned up while working on authentication.
* Improved test coverage further, both with go tests and a forked chef-pedant
  (https://github.com/ctdk/chef-pedant)
* Updated and expanded documentation.

0.3.3
-----
* Data store and indexer tweaks.
* Improved test coverage.

0.3.0
-----
* Added ability to freeze data store and search index to disk.

0.2.1
-----
* Added support for configuration files.
* Fixed issue parsing flags with newer version of go-flags.

0.2.0
-----
* Initial widely announced release. First version with working search.
```

CHAPTER 24

Indices and tables

- genindex
- modindex
- *Search*