



Gate One Documentation

Release 1.2.0

Liftoff Software Corporation

February 05, 2015

CONTENTS

CONTENTS

1.1 About Gate One

Gate One is an open source, web-based terminal emulator with a powerful plugin system. It comes bundled with a plugin that turns Gate One into an amazing SSH client but Gate One can actually be used to run *any* terminal application. You can even embed Gate One into other applications to provide an interface into serial consoles, virtual servers, or anything you like. It's a great supplement to any web-based administration interface.

Gate One works in any browser that supports WebSockets. No browser plugins required!

1.1.1 Licensing

Gate One is released under a dual-license model. You are free to choose the license that best meets your requirements:

- The [GNU Affero General Public License version 3 \(AGPLv3\)](#).
- Gate One Commercial licensing.

Open Source License Requirements

The AGPLv3 (GNU Affero General Public License version 3) is similar to the GPLv3 (GNU General Public License version 3) in that it requires that you publicly release the source code of your application if you distribute binaries that use Gate One. However, it has an additional obligation: You must make your source code available to everyone if your application is provided as Software-as-a-Service (SaaS) or it's part of an Application Service Provider solution.

For example, if you're running Gate One on your server as part of a SaaS solution you must give away all of your source code.

Here are some examples where Open Source licensing makes sense:

- Pre-installing Gate One as part of an open source Linux distribution.
- Embedding Gate One into an open source application that is licensed under the AGPLv3 *or* the GPLv3¹.
- Bundling Gate One with an open source appliance.
- Making Gate One available as part of an open source software repository.

¹ The GPLv3 and AGPLv3 each include clauses (in section 13 of each license) that together achieve a form of mutual compatibility. See [AGPLv3 Section 13](#) and [GPLv3 Section 13](#)

Considerations: Unless you want your source code to be freely available to everyone you should opt for Gate One's Commercial License.

Gate One Commercial Licensing

The Commercial License offerings for Gate One are very flexible and afford businesses the opportunity to include Gate One as part of their products and services without any source code obligations. It also provides licensees with a fully-supported solution and assurances.

Here are some examples where commercial licensing is typically necessary:

- Including Gate One in software sold to customers who install it on their own equipment.
- Selling software that requires the installation of Gate One.
- Selling hardware that comes with Gate One pre-installed.
- Bundling with or including Gate One in any product protected by patents.

Even if you don't plan to embed Gate One into one of your own products, enterprise support options available.

For more information on Gate One Commercial Licensing and support please visit our [website](#).

1.1.2 Prerequisites

Before installing Gate One your system must meet the following requirements:

Requirement	Version
Python	2.6+ or 3.2+
Tornado Framework	2.2+

Note: If using Python 2.6 you'll need to install the ordereddict module: `sudo pip install ordereddict` or you can download it [here](#). As of Python 2.7 OrderedDict was added to the [collections](#) module in the standard library.

The following commands can be used to verify which version of each are installed:

```
user@modern-host:~ $ python -V
Python 2.7.2+
user@modern-host:~ $ python -c "import tornado; print(tornado.version)"
2.4
user@modern-host:~ $
```

Additionally, if you wish to use Kerberos/Active Directory authentication you'll need the [python-kerberos](#) module. On most systems both Tornado and the Kerberos module can be installed with via a single command:

```
user@modern-host:~ $ sudo pip install tornado kerberos
```

...or if you have an older operating system:

```
user@legacy-host:~ $ sudo easy_install tornado kerberos
```

Note: The use of pip is recommended. See <http://www.pip-installer.org/en/latest/installing.html> if you don't have it.

1.1.3 Installation

Gate One can be installed via a number of methods, depending on which package you've got. Assuming you've downloaded the appropriate Gate One package for your operating system to your home directory...

RPM-based Linux Distributions

```
user@redhat:~ $ sudo rpm -Uvh gateone*.rpm
```

Debian-based Linux Distributions

```
user@ubuntu:~ $ sudo dpkg -i gateone*.deb
```

From Source

```
user@whatever:~ $ tar zxvf gateone*.tar.gz; cd gateone*; sudo python setup.py install
```

This translates to: Extract; Change into the gateone* directory; Install.

Tip: You can make your own RPM from the source tarball by executing `sudo python setup.py bdist_rpm` instead of `sudo python setup.py install`.

1.1.4 Configuration

The first time you execute `gateone.py` it will create a default configuration file as `/opt/gateone/settings/10server.conf`:

10server.conf

```
// This is Gate One's main settings file.
{
    // "gateone" server-wide settings fall under "*"
    "*": {
        "gateone": { // These settings apply to all of Gate One
            "address": "",
            "ca_certs": null,
            "cache_dir": "/tmp/gateone_cache",
            "certificate": "certificate.pem",
            "combine_css": "",
            "combine_css_container": "#gateone",
            "combine_js": "",
            "cookie_secret": "Yjg3YmUzOGUxM2Q2NDg3YWU1MTI1YTU3MzVmZTI3YmUzZ",
            "debug": false,
            "disable_ssl": false,
            "embedded": false,
            "enable_unix_socket": false,
            "gid": "0",
            "https_redirect": false,
            "js_init": "",
            "keyfile": "keyfile.pem",
            "locale": "en_US",
            "log_file_max_size": 100000000,
            "log_file_num_backups": 10,
            "log_file_prefix": "/opt/gateone/logs/webserver.log",
            "log_to_stderr": null,
            "logging": "info",
            "origins": [
                "localhost", "127.0.0.1", "enterprise",
                "enterprise.example.com", "10.1.1.100"],
            "pid_file": "/tmp/gateone.pid",
            "port": 443,
            "session_dir": "/tmp/gateone",
            "session_timeout": "5d",
            "syslog_facility": "daemon",
            "syslog_host": null,
            "uid": "0",
            "unix_socket_path": "/tmp/gateone.sock",
            "url_prefix": "/",
            "user_dir": "/opt/gateone/users",
            "user_logs_max_age": "30d"
        }
    }
}
```

Note: The settings under “gateone” can appear in any order.

These options match up directly with Gate One’s command line options which you can view at any time by executing “gateone.py –help”:

```
$ gateone --help
[W 150205 13:58:35 utils:940] Could not import entry point module: gateone.applications.x11
Usage: /usr/local/bin/gateone [OPTIONS]
```


Options:

--help Show this help information

/usr/local/lib/python2.7/dist-packages/tornado/log.py options:

--log_file_max_size max size of log files before rollover (default 100000000)
 --log_file_num_backups number of log files to keep (default 10)
 --log_file_prefix=PATH Path prefix for log files. Note that if you are running multiple tornado processes, log files will be named with the process id (port number)
 --log_to_stderr Send log output to stderr (colored if possible). By default use stderr if --log_file_prefix is set
 --logging=debug|info|warning|error|none Set the Python log level. If 'none', tornado won't touch the logging configuration. (default info)

gateone options:

--address Run on the given address. Default is all addresses (IPv6 included). Multiple addresses can be specified (e.g. --address '127.0.0.1;:::1;10.1.1.100').
 --api_keys The 'key:secret,...' API key pairs you wish to use (only applies if using API authentication)
 --api_timestamp_window How long before an API authentication object becomes invalid. (default 30s)
 --auth Authentication method to use. Valid options are: none, api, cas, google, ssl, kerberos, etc.
 --ca_certs Path to a file containing any number of concatenated CA certificates in PEM format. If not provided, the default is set to 'optional' or 'required'.
 --cache_dir Path where Gate One should store temporary global files (e.g. rendered templates, CSS, etc.)
 --certificate Path to the SSL certificate. Will be auto-generated if not found. (default /etc/gateone/cert.pem)
 --combine_css Combines all of Gate One's CSS Template files into one big file and saves it to the given path.
 --combine_css_container Use this setting in conjunction with --combine_css if the <div> where Gate One lives is in a container.
 --combine_js Combines all of Gate One's JavaScript files into one big file and saves it to the given path.
 --command DEPRECATED: Use the 'commands' option in the terminal settings.
 --config DEPRECATED. Use --settings_dir. (default /opt/gateone/server.conf)
 --configure Only configure Gate One (create SSL certs, conf.d, etc). Do not start any Gate One processes.
 --cookie_secret Use the given 45-character string for cookie encryption.
 --debug Enable debugging features such as auto-restarting when files are modified. (default False)
 --disable_ssl If enabled Gate One will run without SSL (generally not a good idea). (default False)
 --embedded When embedding Gate One this option is available to plugins, applications, and templates. (default False)
 --enable_unix_socket Enable Unix socket support. (default False)
 --gid Drop privileges and run Gate One as this group/gid. (default 1000)
 --https_redirect If enabled a separate listener will be started on port 80 that redirects users to the https listener.
 --js_init A JavaScript object (string) that will be used when running GateOne.init() inside index.html. (default GateOne.init({theme: 'white'}))
 --keyfile Path to the SSL keyfile. Will be auto-generated if none is provided. (default /etc/gateone/key.pem)
 --locale The locale (e.g. pt_PT) Gate One should use for translations. If not provided, will default to en_US.
 --multiprocessing_workers The number of processes to spawn use when using multiprocessing. Default is: <number of processors>
 --new_api_key Generate a new API key that an external application can use to embed Gate One. (default False)
 --origins A semicolon-separated list of origins you wish to allow access to your Gate One server. (e.g. http://foo.com;foo.bar;) and IP addresses. This value must contain all the hostnames/IPs that you wish to allow access to. If not specified to allow access from anywhere. NOTE: Using a '*' is only a good idea if you are running Gate One on localhost;127.0.0.1;enterprise;127.0.1.1)
 --pam_realm Basic auth REALM to display when authenticating clients. Default: hostname. Only relevant if PAM authentication is enabled.
 --pam_service PAM service to use. Defaults to 'login'. Only relevant if PAM authentication is enabled.
 --pid_file Define the path to the pid file. (default /home/riskable/.gateone/gateone.pid)
 --port Run on the given port. (default 10443)
 --session_dir Path to the location where session information will be stored. (default /home/riskable/.gateone/sessions)
 --session_timeout Amount of time that a session is allowed to idle before it is killed. Accepts <num>X where X is minutes, hours, and days. Set to '0' to disable the ability to resume sessions. (default 5d)
 --settings_dir Path to the settings directory. (default /home/riskable/.gateone/conf.d)

<code>--ssl_auth</code>	Enable the use of client SSL (X.509) certificates as a secondary authentication factor of 'none', 'optional', or 'required'. NOTE: Only works if the 'ca_certs' option is co
<code>--sso_realm</code>	Kerberos REALM (aka DOMAIN) to use when authenticating clients. Only relevant if Kerb
<code>--sso_service</code>	Kerberos service (aka application) to use. Only relevant if Kerberos authentication i
<code>--syslog_facility</code>	Syslog facility to use when logging to syslog (if <code>syslog_session_logging</code> is enabled). local3, local4, local5, local6, local7, lpr, mail, news, syslog, user, uucp. (default
<code>--uid</code>	Drop privileges and run Gate One as this user/uid. (default 1000)
<code>--unix_socket_path</code>	Path to the Unix socket (if <code>--enable_unix_socket=True</code>). (default <code>/tmp/gateone.sock</code>)
<code>--url_prefix</code>	An optional prefix to place before all Gate One URLs. e.g. <code>/gateone/</code> . Use this if C be located at some sub-URL path. (default <code>/</code>)
<code>--user_dir</code>	Path to the location where user files will be stored. (default <code>/home/riskable/.gateone</code>)
<code>--user_logs_max_age</code>	Maximum length of time to keep any given user log before it is automatically removed.
<code>--version</code>	Display version information.

terminal options:

<code>--detach</code>	Wrap terminals with <code>dtach</code> . Allows sessions to be resumed even if Gate One is stopped a
<code>--kill</code>	Kill any running Gate One terminal processes including <code>dtach</code> 'd processes. (default Fal
<code>--session_logging</code>	If enabled, logs of user sessions will be saved in <code><user_dir>/<user>/logs</code> . Default: F
<code>--syslog_session_logging</code>	If enabled, logs of user sessions will be written to syslog. (default False)

Commands:

Usage: `/usr/local/bin/gateone <command> [OPTIONS]`

GateOne supports many different CLI 'commands' which can be used to invoke special functionality provided by plugins and their own options and most will have a `--help` function of their own.

Commands provided by 'gateone':

<code>broadcast</code>	Broadcast messages to users connected to Gate One.
<code>install_license</code>	Install a commercial license for Gate One.
<code>validate_authobj</code>	Test API authentication by validating a JSON auth object.

Commands provided by 'gateone.applications.terminal':

<code>termlog</code>	View terminal session logs.
----------------------	-----------------------------

Example command usage:

`/usr/local/bin/gateone termlog --help`

These options are detailed below in the format of:

Name

`--command_line_option`

Default value as it is defined in the `.conf` files in the `'settings_dir'`.

Description

Tornado Framework Options

The options below are built-in to the Tornado framework. Since Gate One uses Tornado they'll always be present.

log_file_max_size

--log_file_max_size=bytes

```
"log_file_max_size": 104857600 // Which is the result of: 100 * 1024 * 1024
```

This defines the maximum size of Gate One's web server log file in bytes before it is automatically rotated.

Note: Web server log settings don't apply to Gate One's user session logging features.

log_file_num_backups

--log_file_num_backups=integer

```
"log_file_max_size": 10
```

The maximum number of backups to keep of Gate One's web server logs.

log_file_prefix

--log_file_prefix=string (file path)

```
"log_file_prefix": "/opt/gateone/logs/webserver.log"
```

This is the path where Gate One's web server logs will be kept.

Note: If you get an error like this:

```
IOError: [Errno 13] Permission denied: '/opt/gateone/logs/webserver.log'
```

It means you need to change this setting to somewhere your user has write access such as `/var/tmp/gateone_logs/webserver.log`.

log_to_stderr

--log_to_stderr=boolean

```
"log_to_stderr": False
```

This option tells Gate One to send web server logs to stderr (instead of to the log file).

logging

--logging=string (info|warning|error|none)

```
"logging": "info"
```

Specifies the log level of the web server logs. The default is "info". Can be one of, "info", "warning", "error", or "none".

Global Gate One Options

The options below represent settings that are specific to Gate One, globally. Meaning, they're not tied to specific applications or plugins.

address

--address=string (IPv4 or IPv6 address)

`"address": ""` // Empty string means listen on all addresses

Specifies the IP address or hostname that Gate One will listen for incoming connections. Multiple addresses may be provided using a semicolon as the separator. For example:

`"address": "localhost:::1;10.1.1.100"` // Listen on localhost, the IPv6 loopback address, and 10.1.1.100

See also:

`--port`

api_keys

--api_keys=string (key1:secret1,key2:secret2,...)

```
"api_keys": {  
  "ZWVkoWJjZ23yNjNlNDQ1YWE3MThiYmI0M72suJFhODFiZ": "NTg5NTl0OTIyMD1lNGU1MzkzZDM4NjVhZWNGNDdln2RmO"  
}
```

Specifies a list of API keys (key:value pairs) that can be used when performing using Gate One's authentication API.

api_timestamp_window

--api_timestamp_window=string (special: [0-9]+[smhd])

`"api_timestamp_window": "30s"`

This setting controls how long API authentication objects will last before they expire if `--auth` is set to 'api' (default is 30 seconds). It accepts the following <num><character> types:

Character	Meaning	Example
s	Seconds	'60s' = 60 Seconds
m	Minutes	'5m' = 5 Minutes
h	Hours	'24h' = 24 Hours
d	Days	'7d' = 7 Days

Note: If the value is too small clock drift between the Gate One server and the web server embedding it can cause API authentication to fail. If the setting is too high it provides a greater time window in which an attacker can re-use that token in the event the Gate One server is restarted. **Important:** Gate One keeps track of used authentication objects but only in memory. If the server is restarted there is a window in which an API authentication object can be re-used (aka an authentication replay attack). That is why you want the `api_timestamp_window` to be something short but not too short as to cause problems if a clock gets a little out of sync.

auth

`--auth=string (none|pam|google|kerberos|api)`

`"auth": "none"`

Specifies how you want Gate One to authenticate clients. One of, "none", "pam", "google", "kerberos", or "api".

ca_certs

`--ca_certs=string (file path)`

`"ca_certs": "/opt/gateone/ca_certs.pem" // Default is None`

Path to a file containing any number of concatenated CA certificates in PEM format. They will be used to authenticate clients if the `--ssl_auth` option is set to 'optional' or 'required'.

cache_dir

`--cache_dir=string (directory path)`

`"cache_dir": "/tmp/gateone_cache"`

Path to a directory where Gate One will cache temporary information (for performance/memory reasons). Mostly for things like templates, JavaScript, and CSS files that have been rendered and/or minified.

certificate

`--certificate=string (file path)`

`"certificate": "/etc/gateone/ssl/certificate.pem"`

The path to the SSL certificate Gate One will use in its web server.

Note: The file must be in PEM format.

combine_css

`--combine_css=string (file path)`

This option tells Gate One to render and combine all its CSS files into a single file (i.e. so you can deliver it via some other web server/web cache/CDN). The file will be saved at the provided path.

combine_css_container

`--combine_css_container=string (e.g. 'gateone')`

When combining CSS into a single file, this option specifies the name of the '#gateone' container element (if something else). It is used when rendering CSS templates.

combine_js

`--combine_js=string (file path)`

This option tells Gate One to combine all its JavaScript files into a single file (i.e. so you can deliver it via some other web server/web cache/CDN). The file will be saved at the provided path.

command

`--command=string (program path)`

Deprecated since version 1.2: This option has been replaced by the Terminal application's 'commands' option which supports multiple. See *Terminal Configuration*.

config

`--config=string (file path)`

Deprecated since version 1.2: This option has been replaced by the `--settings_dir` option.

cookie_secret

`--cookie_secret=string ([A-Za-z0-9])`

`"cookie_secret": "A45CharacterStringGeneratedAtRandom012345678" // NOTE: Yours will be different ;)`

This is a 45-character string that Gate One will use to encrypt the cookie stored at the client. By default Gate One will generate one at random when it runs for the first time. Only change this if you know what you're doing.

Note: If you change this string in the `10server.conf` you'll need to restart Gate One for the change to take effect.

What happens if you change it

All users existing, unexpired sessions will need to be re-authenticated. When this happens the user will be presented with a dialog box that informs them that the page hosting Gate One will be reloaded automatically when they click "OK".

Tip: You may have to change this key at a regular interval throughout the year depending on your compliance needs. Every few months is probably not a bad idea regardless.

debug

`--debug=boolean`

`"debug": false`

Turns on Tornado's debug mode: If a change is made to any Python code while **gateone.py** is running it will automatically restart itself. Cached templates will also be regenerated.

disable_ssl

`--disable_ssl=boolean`

`"disable_ssl": false`

Disables SSL support in Gate One. Generally not a good idea unless you know what you're doing. There's really only two reasons why you'd want to do this:

- Gate One will be running behind a proxy server that handles the SSL encryption.
- Gate One will only be connected to via localhost (kiosks, console devices, etc).

embedded

`--embedded=boolean`

`"embedded": false`

This option is available to applications, plugins, and CSS themes/templates (if desired). It is unused by Gate One proper but can be used at your discretion when embedding Gate One.

Note: This isn't the same thing as "embedded mode" in the JavaScript code. See *GateOne.prefs.embedded* in *gateone.js*.

enable_unix_socket

`--enable_unix_socket=boolean`

`"enable_unix_socket": false`

Tells Gate One to listen on a [Unix socket](#). The path to said socket is defined in `--unix_socket_path`.

gid

`--gid=string`

`"gid": "0" // You could also put "root", "somegroup", etc`

If run as root, Gate One will drop privileges to this group/gid after starting up. Default: 0 (aka root)

https_redirect

`--https_redirect`

`"https_redirect": false`

If `https_redirect` is enabled, Gate One will listen on port 80 and redirect incoming connections to Gate One's configured port using HTTPS.

js_init

--js_init=string (JavaScript Object)

```
"js_init": ""
```

This option can be used to pass parameters to the `GateOne.init()` function whenever Gate One is opened in a browser. For example:

```
"js_init": "{\"theme\": 'white', 'fontSize': '120%'}"
```

For a list of all the possible options see `GateOne.prefs` in the *Developer Documentation* under *gateone.js*.

Note: This setting will only apply if you're *not* using embedded mode.

keyfile

--keyfile=string (file path)

```
"keyfile": "/etc/gateone/ssl/keyfile.pem"
```

The path to the SSL key file Gate One will use in its web server.

Note: The file must be in PEM format.

locale

--locale=string (locale string)

```
"locale": "en_US"
```

This option tells Gate One which local translation (native language) to use when rendering strings. The first time you run Gate One it will attempt to automatically detect your locale using the `$LANG` environment variable. If this variable is not set it will fall back to using `en_US`.

Note: If no translation exists for your local the English strings will be used.

new_api_key

--new_api_key

This command line option will generate a new, random API key and secret for use with applications that will be embedding Gate One. Instructions on how to use API-based authentication can be found in the *Embedding Gate One Into Other Applications*.

Note: By default generated API keys are placed in `<settings_dir>/20api_keys.conf`.

origins

--origins=string (semicolon-separated origins)


```
"origins": ["localhost", "127.0.0.1", "enterprise", "enterprise\\.*.com", "10.1.1.100"]
```

Note: The way you pass origins on the command line is very different from how they are stored in 10server.conf. The CLI option uses semicolons to delineate origins whereas the 10server.conf contains an actual JSON array.

By default Gate One will only allow connections from web pages that match the configured origins. If a user is denied access based on a failed origin check a message will be logged like so:

```
[I 120831 15:32:12 gateone:1043] WebSocket closed (ANONYMOUS).  
[E 120831 15:32:17 gateone:943] Access denied for origin: https://somehost.company.com
```

Note: Origins do not contain protocols/schemes, paths, or trailing slashes!

Warning: If you see unknown origins the logs it could be an attacker trying to steal your user's sessions! The origin that appears in the log will be the hostname that was used to connect to the Gate One server. This information can be used to hunt down the attacker. Of course, it could just be that a new IP address or hostname has been pointed to your Gate One server and you have yet to add it to the `--origins` setting.

pam_realm

`--pam_realm=string (hostname)`

```
"pam_realm": "somehost"
```

If `--auth` is set to "pam" Gate One will present this string in the BASIC auth dialog (essentially, the login dialog will say, "REALM: <pam_realm>"). Also, the user's directory will be created in `--user_dir` as `user@`. Make sure to use path-safe characters!

pam_service

`--pam_service=string`

```
"pam_service": "login"
```

If `--auth` is set to "pam", tells Gate One which PAM service to use when authenticating clients. Defaults to 'login' which is typically configured via `/etc/pam.d/login`.

Tip: You can change this to "gateone" and create a custom PAM config using whatever authentication back-end you want. Just set it as such and create `/etc/pam.d/gateone` with whatever PAM settings you like.

pid_file

`--pid_file=string`

```
"pid_file": "/var/run/gateone.pid"
```

The path to Gate One's `PID` file.

Note: If you're not running Gate One as root it's possible to get an error like this:

```
IOError: [Errno 13] Permission denied: '/var/run/gateone.pid'
```

This just means you need to change this setting to point somewhere your user has write access such as `/tmp/gateone.pid`.

port

`--port=integer (1-65535)`

```
"port": 443
```

The port Gate One should listen for connections.

Note: Gate One must started as root to utilize ports 1-1024.

Tip: If you set `--uid` and/or `--gid` to something other than "0" (aka root) Gate One will drop privileges to that user/group after it starts. This will allow the use of ports under 1024 while maintaining security best practices by running as a user/group with lesser privileges.

session_dir

`--session_dir=string (file path)`

```
"session_dir": "/tmp/gateone"
```

The path where Gate One should keep temporary user session information. Defaults to `/tmp/gateone` (will be auto-created if not present).

session_timeout

`--session_timeout=string (special: [0-9]+[smhd])`

```
"session_timeout": "5d"
```

This setting controls how long Gate One will wait before force-killing a user's disconnected session (i.e. where the user hasn't used Gate One in, say, "5d"). It accepts the following `<num><character>` types:

Character	Meaning	Example
s	Seconds	'60s' □ 60 Seconds
m	Minutes	'5m' □ 5 Minutes
h	Hours	'24h' □ 24 Hours
d	Days	'7d' □ 7 Days

Note: Even if you're using `--detach` all programs associated with the user's session will be terminated when the timeout is reached.

ssl_auth

`--ssl_auth=string (None|optional|required)`

```
"ssl_auth": "none"
```

If set to 'required' or 'optional' this setting will instruct Gate One to authenticate client-side SSL certificates. This can be an excellent added layer of security on top of Gate One's other authentication options. Obviously, only the 'required' setting adds this protection. If set to 'optional' it merely adds information to the logs.

Note: This option must be set to 'required' if `--auth` is set to "ssl". The two together allow you to use SSL certificates as a single authentication method.

sso_realm

`--sso_realm=string (Kerberos realm or Active Directory domain)`

```
"sso_realm": "EXAMPLE.COM"
```

If `--auth` is set to "kerberos", tells Gate One which Kerberos realm or Active Directory domain to use when authenticating users. Otherwise this setting will be ignored.

Note: SSO stands for Single Sign-On.

sso_service

`--sso_service=string (kerberos service name)`

```
"sso_service": "HTTP"
```

If `--auth` is set to "kerberos", tells Gate One which Kerberos service to use when authenticating clients. This is the 'service/' part of a principal or servicePrincipalName (e.g. **HTTP**/somehost.example.com).

Note: Unless you *really* know what you're doing do not use anything other than HTTP (in all caps).

syslog_facility

`--syslog_facility=string (auth|cron|daemon|kern|local0|local1|local2|local3|local4|local5|local6|local7|lpr|n`

```
"syslog_facility": "daemon"
```

if `--syslog_session_logging` is set to True, specifies the syslog facility that user session logs will use in outgoing syslog messages.

uid

`--uid=string`

```
"uid": "0" // You could also put "root", "someuser", etc
```

If run as root, Gate One will drop privileges to this user/uid after starting up. Default: 0 (aka root)

unix_socket_path

`--unix_socket_path=string (file path)`

`"unix_socket_path": "/tmp/gateone.sock"`

Path to the Unix socket (if `--enable_unix_socket` is "true").

url_prefix

`--url_prefix=string (e.g. ''/ssh/'')`

`"url_prefix": "/"`

This specifies the URL path Gate One will live when it is accessed from a browser. By default Gate One will use `"/"` as its base URL; this means that you can connect to it using a URL like so: <https://mygateone.company.com/>

That `"/"` at the end of the above URL is what the `url_prefix` is specifying. If you wanted your Gate One server to live at <https://mygateone.company.com/gateone/> you could set `url_prefix="/gateone/"`.

Note: This feature was added for users running Gate One behind a reverse proxy so that many apps (like Gate One) can all live behind a single base URL.

Tip: If you want to place your Gate One server on the Internet but don't want it to be easily discovered/enumerated you can specify a random string as the gateone prefix like `url_prefix="/fe34b0e0c074f486c353602/"`. Then only those who have been made aware of your obfuscated URL will be able to access your Gate One server (at <https://gateone.company.com/fe34b0e0c074f486c353602/>)

user_dir

`--user_dir=string (file path)`

`"user_dir": "/var/lib/gateone/users"`

Specifies the location where persistent user files will be kept. Things like session logs, ssh files (known_hosts, keys, etc), and similar are stored here.

user_logs_max_age

`--user_logs_max_age=string (special: [0-9]+[smhd])`

`"user_dir": "30d"`

This setting controls how long Gate One will wait before old user session logs are cleaned up. It accepts the following `<num><character>` types:

Character	Meaning	Example
s	Seconds	'60s' = 60 Seconds
m	Minutes	'5m' = 5 Minutes
h	Hours	'24h' = 24 Hours
d	Days	'7d' = 7 Days

Note: This is *not* a Terminal-specific setting. Other applications can and will use user session logs directory (<user_dir>/logs).

version

--version

Displays the current Gate One version as well as the version information of any installed applications. Example:

```
root@host:~ $ gateone --version
Gate One:
  Version: 1.2.0 (20140226213756)
Installed Applications:
  Terminal Version: 1.2
  X11 Version: 1.0
```

Terminal Application Options

The options below are specific (and supplied by) the Terminal application.

dtach

--dtach=boolean

"dtach": true

This feature is special: It enables Gate One to be restarted (think: upgraded) without losing user's connected sessions. This option is enabled by default.

If dtach support is enabled but the dtach command cannot be found Gate One will output a warning message in the log.

Note: If you ever need to restart Gate One (and dtach support is enabled) users will be shown a message indicating that they have been disconnected and their browsers should automatically reconnect in 5 seconds. A 5-second maintenance window ain't bad!

kill

--kill

If running with dtach support, this will kill all user's running terminal applications. Giving everyone a fresh start, as it were.

session_logging

--session_logging

session_logging = True

This tells Gate One to enable server-side logging of user terminal sessions. These logs can be viewed or played back (like a video) using the *Log Viewer* application.

Note: Gate One stores logs of user sessions in the location specified in the `--user_dir` option.

syslog_session_logging

--syslog_session_logging

"syslog_session_logging": `false`

This option tells Gate One to send logs of user sessions to the host's syslog daemon. Special characters and escape sequences will be sent as-is so it is up to the syslog daemon as to how to handle them. In most cases you'll wind up with log lines that look like this:

```
Oct  1 19:18:22 gohost gateone: ANONYMOUS 1: Connecting to: ssh://user@somehost:22
Oct  1 19:18:22 gohost gateone: ANONYMOUS 1: #033]0;user@somehost#007
Oct  1 19:18:22 gohost gateone: ANONYMOUS 1: #033]_;ssh|user@somehost:22#007
```

Note: This option enables centralized logging if your syslog daemon is configured to use a remote log host.

1.2 Gate One Applications

Gate One is more than just a web-based terminal. It can be host to many other applications as well. These applications have their own documentation...

1.2.1 Terminal

Gate One's Terminal application

User Guide

The Interface

When you first connect to a Gate One terminal you should see something like this:

Note: The text zoom level was increased for all of these screenshots to make them easier to read.

Detailed Interface Overview

Here's an overview of what each GUI element does:

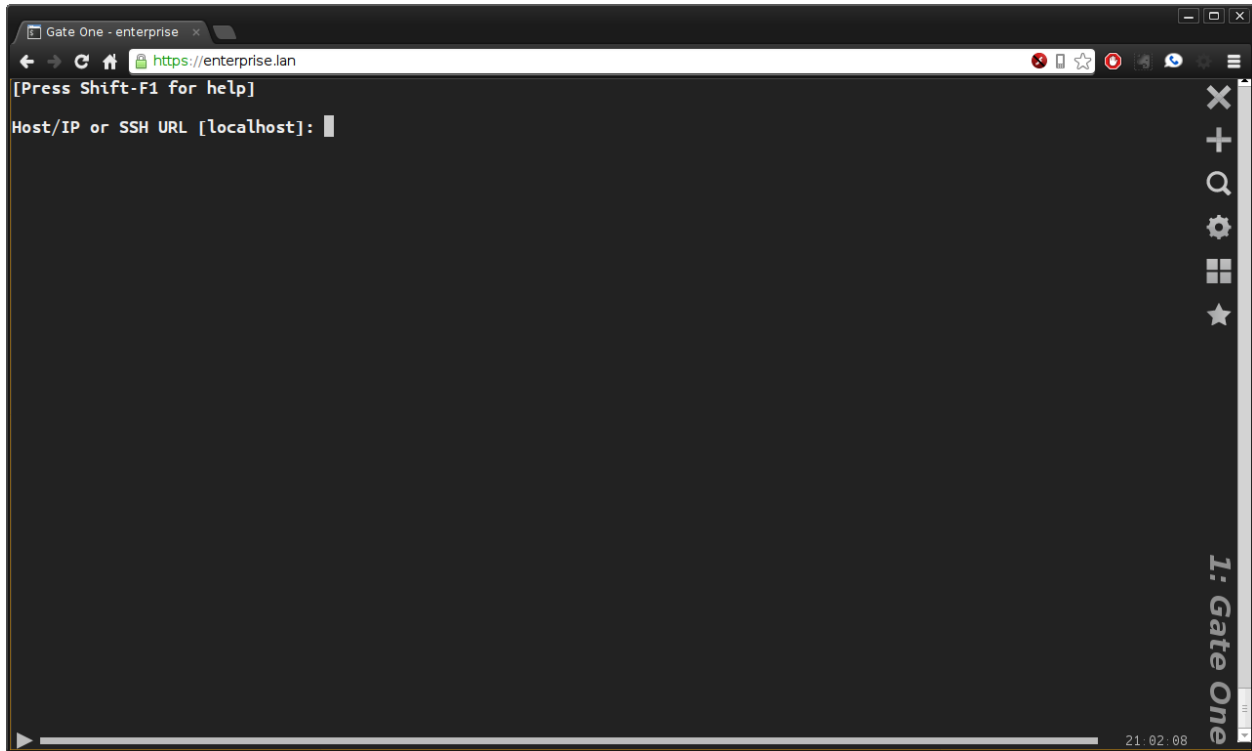


Figure 1.1: The default login screen with the SSH plugin enabled.

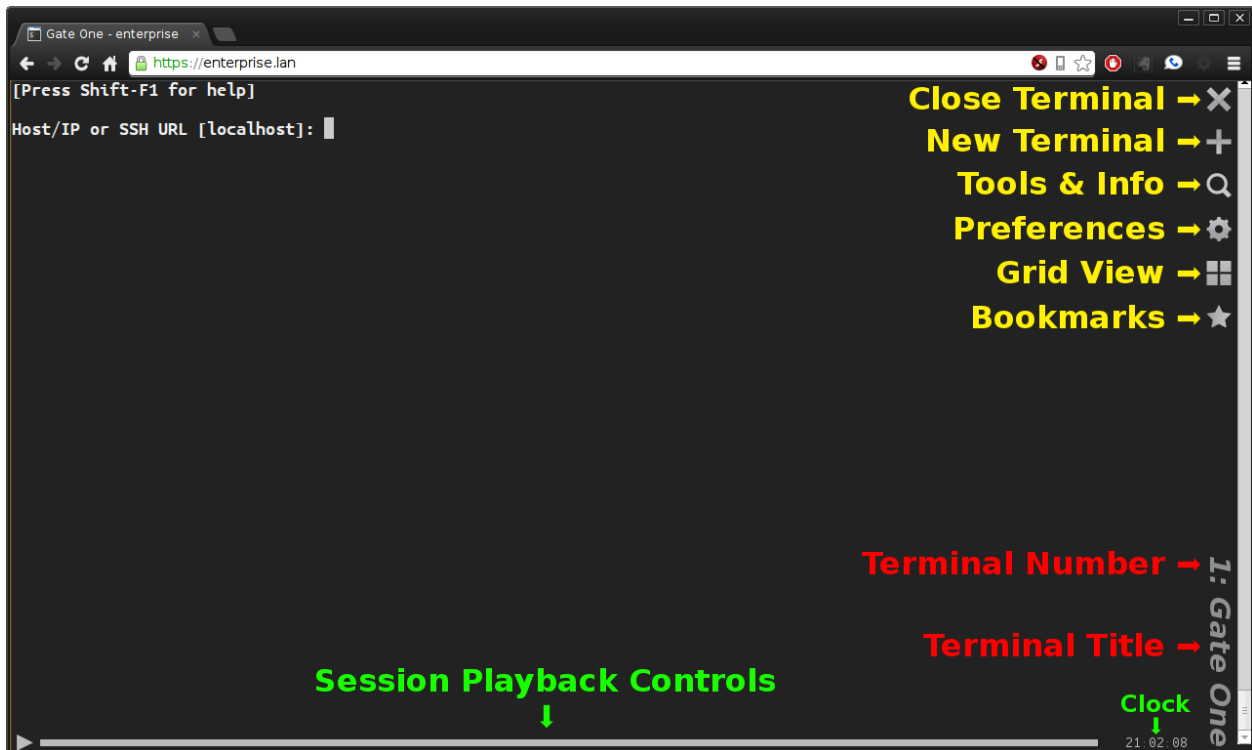


Figure 1.2: Gate One's interface, explained

Keyboard Shortcuts

Function	Shortcut
View The Help	Shift-F1
Open Terminal	Control-Alt-N
Close Terminal	Control-Alt-W
Show Grid	Control-Alt-G
Switch to the terminal on the left	Shift-LeftArrow
Switch to the terminal on the right	Shift-RightArrow
Switch to the terminal above	Shift-UpArrow
Switch to the terminal below	Shift-DownArrow

The Toolbar

Gate One's toolbar consists of a number of icons at the right of the window. These icons are the main source of interaction with Gate One. It should also be noted that all of Gate One's icons are inline SVG graphics... So they will scale with your font size and will change color based on your CSS theme. Another nice feature of the toolbar is that it's trivial for plugins to add their own icons.

Note: Why are the icons and title on the right and not the top? Because these days monitors are much wider than they are tall. In other words, vertical screen space is at a premium while there's plenty of room on the sides.

Close Terminal and New Terminal These icons do precisely what you'd expect from a modern GUI: The X icon closes the terminal in the current view while the + icon opens a new terminal.

Note: If you close the last open terminal a new one will be opened.

Terminal Info and Tools Gate One's information panel is the place to get information about the current terminal. By default, it will display the current terminal title, how long the terminal has been open, and the number of rows and columns. It also presents some options to the user which are outlined below...

Tip: You can manually change the current terminal title by clicking on it.

Log Viewer Gate One's log viewer provides a mechanism for viewing the logs of terminal sessions stored on the server. Log metadata as well as a preview can be viewed by simply clicking on any given log. Playback and flat (traditional) viewing options are also available. These will open in a new window.

Tip: When you open the log viewer it will display a message indicating how many logs there are associated with your user account along with the total amount of space the logs are taking up on the server.

Note: Gate One's log format is pre-compressed using gzip. There's no need to compress them.

Export Current Session When this button is clicked it will open up a new browser tab that will play back the current terminal's session. This recording is self-contained and can be saved to your computer for playback later. Everything needed to play back the recording is contained within the HTML file itself.



Figure 1.3: This is what you see when you click on the magnifying glass icon

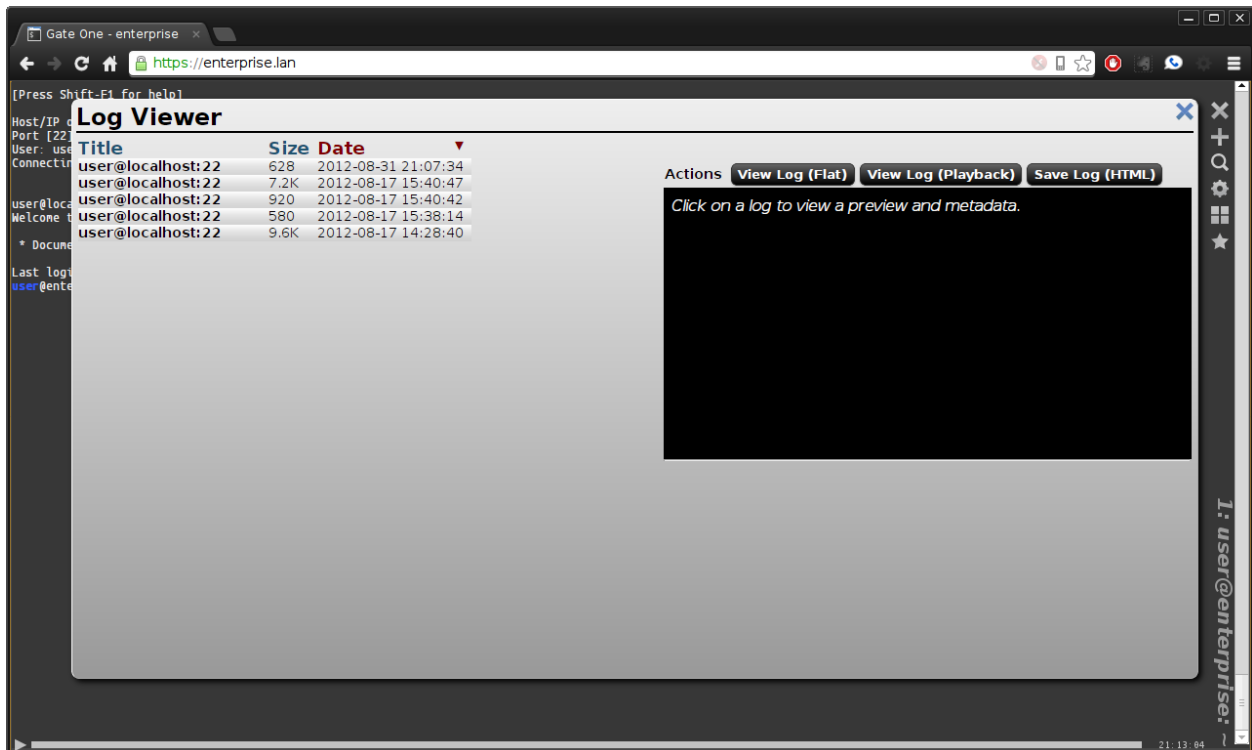


Figure 1.4: Gate One's Log Viewer (unzoomed)

You can share it with friends, plop it into an iframe on a website, or just email it to someone. It will even auto-scale itself (down) if necessary to fit within the current frame or window.

Note: This kind of session recording is merely a shortcut to quickly exporting the current terminal session. You can always access your server-side session logs from within the log viewer.

Monitor for Activity/Inactivity This feature allows you to monitor the current terminal for either activity (e.g. something changes) or inactivity (e.g. when the terminal stops changing). When either of these events is triggered Gate One will play a sound and pop up an alert to notify you which terminal has passed the threshold for activity or inactivity.

Tip: This feature is very handy for when you want to know when, say, a download is complete (inactivity: “wget <url>” finishes) or when someone hits your website (activity: a “tail -f” on the log suddenly has output).

SSH Plugin: Duplicate Session You’ll see this option if the SSH plugin is enabled... This button allows you to duplicate your current SSH session. It will open a new SSH connection to the current server using the exact same SSH connect string (e.g. `ssh://user@host:22`) that was used to connect originally. Also, if possible, it will utilize the existing SSH tunnel for this connection which means you won’t have to re-enter your password. When this (awesome) feature is invoked you’ll see a message indicating as such in the terminal:

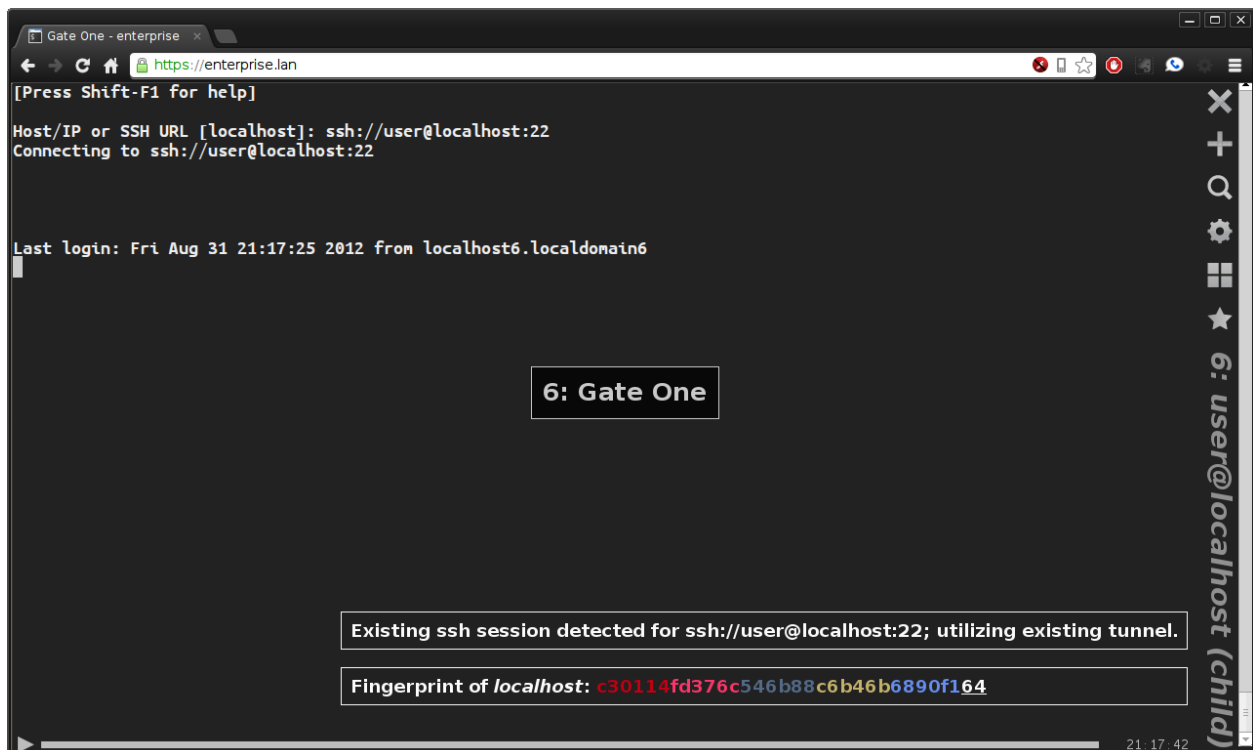


Figure 1.5: After duplicating an SSH session. No password required!

SSH Plugin: Manage Identities The SSH plugin includes an interface for managing all of your SSH identities (aka SSH keys). Here, SSH identities (private and public key files) can be generated, downloaded,

uploaded, or deleted. There is also support for uploading (or replacing existing) X.509 certificates that may be associated with a given identity. X.509 support is important because it provides the ability for keys to be revoked (e.g. in the event that an employee leaves your company). X.509 certificates can also restrict what privileges a user has when logging into a server via SSH (e.g. disallowing port forwarding). If any of these restrictions are present in a given Identity's X.509 certificate they will be displayed in place of the randomart field.

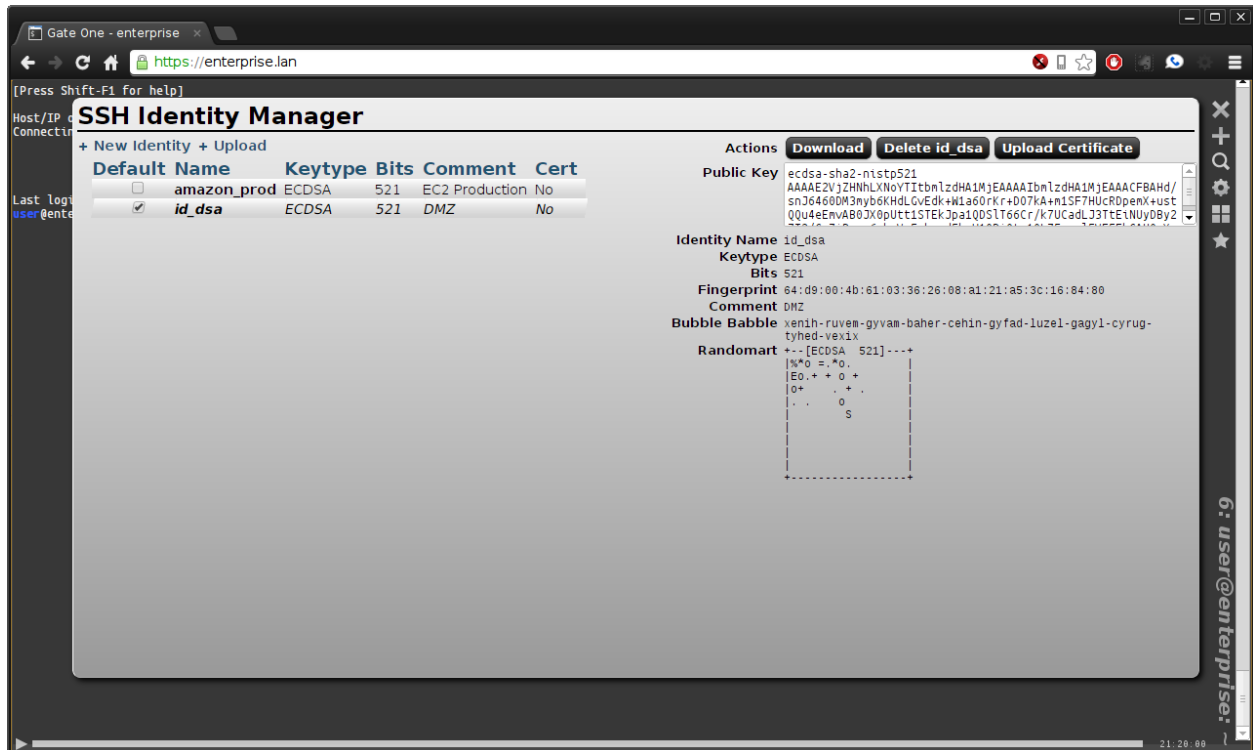


Figure 1.6: SSH Identity Manager (unzoomed)

Tip: If you hover your mouse over the title of each column it will provide detailed description of what it means.

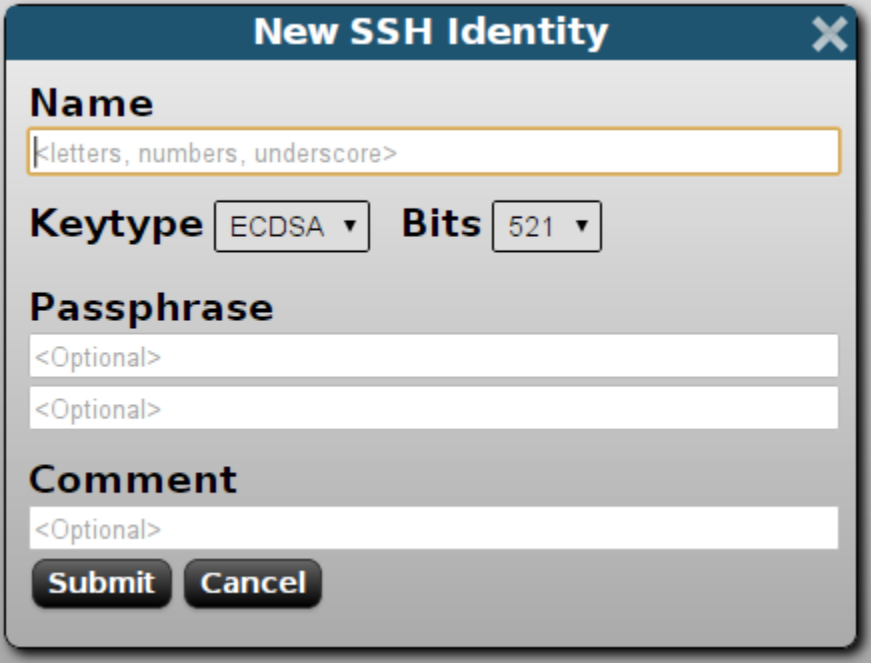
The SSH Identity Manager also allows you to generate new identities...

Upload existing identities...

Tip: If you upload a private key without a public key Gate One will automatically derive the public key from the private one. Super handy for Amazon EC2 SSH identities where they don't give you the public key.


And upload X.509 certificates...

Edit Known Hosts Clicking this button will bring up an editor for Gate One's equivalent of `~/ssh/known_hosts` (same file, different location). This will be handy if some server you connect to on a regular basis ever changes its host key... You'll need to delete the corresponding line.



The 'New SSH Identity' dialog box features a dark blue header with a close button. It contains four main sections: 'Name' with a text input field showing a placeholder '<letters, numbers, underscore>'; 'Keytype' with a dropdown menu set to 'ECDSA' and 'Bits' with a dropdown menu set to '521'; 'Passphrase' with two stacked text input fields, both showing '<Optional>'; and 'Comment' with a single text input field showing '<Optional>'. At the bottom are 'Submit' and 'Cancel' buttons.

Figure 1.7: Generate New Identity



The 'Upload SSH Identity' dialog box has a dark blue header with a close button. It contains three sections for file uploads: 'Private Key', 'Optional: Public Key', and 'Optional: Certificate'. Each section has a 'Choose File' button and a text area showing 'No file chosen'. Below these sections is a 'NOTE' stating: 'If a public key is not provided one will be automatically generated using the private key. You may be asked for the passphrase to perform this operation.' At the bottom are 'Submit' and 'Cancel' buttons.

Figure 1.8: Upload Identity

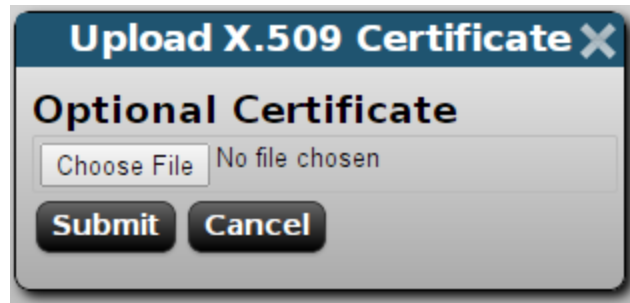


Figure 1.9: Upload X.509 Certificate

Note: Line numbers in the textarea are forthcoming (to make finding the appropriate host line easier).

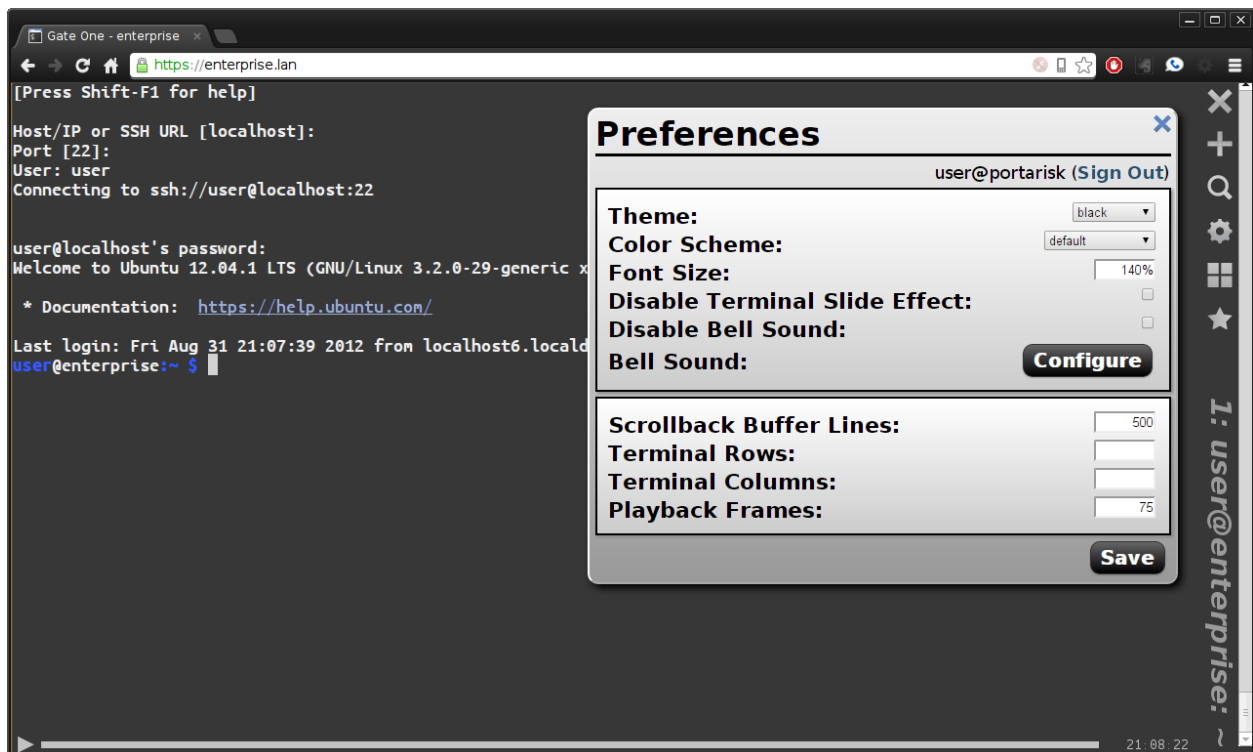


Figure 1.10: Gate One's Settings Panel

The Settings Panel These options are detailed below...

Theme This controls the look and feel of Gate One. When selected, the chosen theme will take effect right away.

Tip: Themes are just CSS files and are easy to edit. `black.css` and `white.css` are in `<path>/templates/themes`. Copy one and start making your own! If it turns out pretty good send it to us and we'll include it with Gate One.

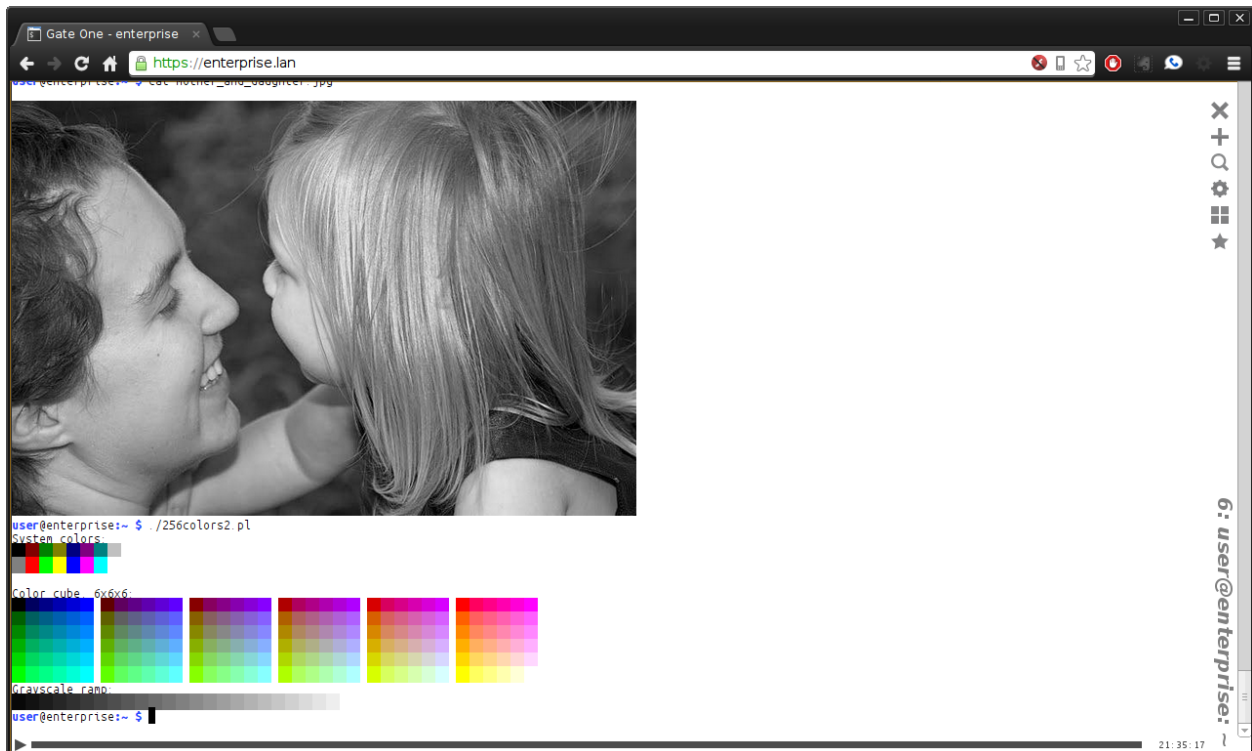


Figure 1.11: White Theme

Note: The black scheme doesn't actually have a black background (it's #222)... Why? So the panels can have shadows which provides important contrast. Essentially, it is easier on the eyes.

Color Scheme This is similar to the “Theme” option above but it only controls the colors of terminal text (aka renditions).

Note: CSS color schemes can be found in <path to gateone>/templates/term_colors.

Scrollbar Buffer Lines This option tells Gate One how many lines to keep in the scrollbar buffer (in memory). When you're typing or when a terminal is updating itself Gate One only updates the browser window with what falls within the terminal's rows and columns. Only after a timeout of 3.5 seconds does it re-attach the scrollbar buffer. When this happens the browser has to render all that text; the more there is the longer it takes (milliseconds). Even on a slow system 500 lines (the default) should be unnoticeably speedy.

Tip: You don't have to wait for the 3.5 second timeout: Just start scrolling and the timeout will be cancelled and the scrollbar buffer will be immediately prepended to the current view.

Note: Why the complexity? The more text that is being rendered, the slower the browser will be able to update your terminal window. If we updated the current number of rows + the number of lines in the scrollbar buffer every time you pressed a key this would quickly bog down your browser and make Gate One considerably less responsive.

Playback Frames This option controls how many frames of real-time session playback will be kept in working memory. The higher the number, the more memory it will use. Also, the more terminals you have open the higher the memory use as well. Having said that, 200-500 frames per terminal shouldn't be of any concern for a modern computer.

Tip: If you hold down the Shift key while scrolling with your mouse wheel it will move backwards and forwards in the playback buffer instead of scrolling up and down. It is a handy way to see the history of full-screen applications such as 'top'.

Terminal Rows and Terminal Columns By default these are blank which means Gate One will automatically figure out how many rows and columns will fit in a given terminal window. If you set these, Gate One will force these values on all running terminal programs. The ability to set this on a per-terminal basis is forthcoming.

Note: Why would anyone bother? Some legacy/poorly-written terminal programs only work properly in a terminal window of 24 rows and 80 columns.

Gate One's Grid Gate One lays out terminals in a grid like so:

Terminal 1	Terminal 2
Terminal 3	Terminal 4
Terminal 5	Terminal 6
So on	And so on

The grid view can be invoked by either clicking on the Grid icon (four squares) in the toolbar or via the Ctrl-Alt-G keyboard shortcut. Here's what it looks like:

The Bookmark Manager The first time you open the Bookmarks manager it will be empty:

Bookmarks can be added by clicking on "New":

Here's an example of the new bookmark form, filled out with a new SSH bookmark:

After submitting the form (which doesn't actually submit anything to the Gate One server) we can see our first bookmark in the panel:

Here's what the panel will look like after you've added a number of bookmarks:

Tip: Clicking on any of those tags will filter the current view to only show bookmarks that have been tagged as such.

Lastly, here's what happens when you click on an SSH bookmark:

In the example above, the ssh:// URL was automatically entered for us. All we had to do was enter our password.

Terminal Developer Documentation

Python Code

app_terminal.py - Gate One Terminal Application A Gate One Application (GOApplication) that provides a terminal emulator.

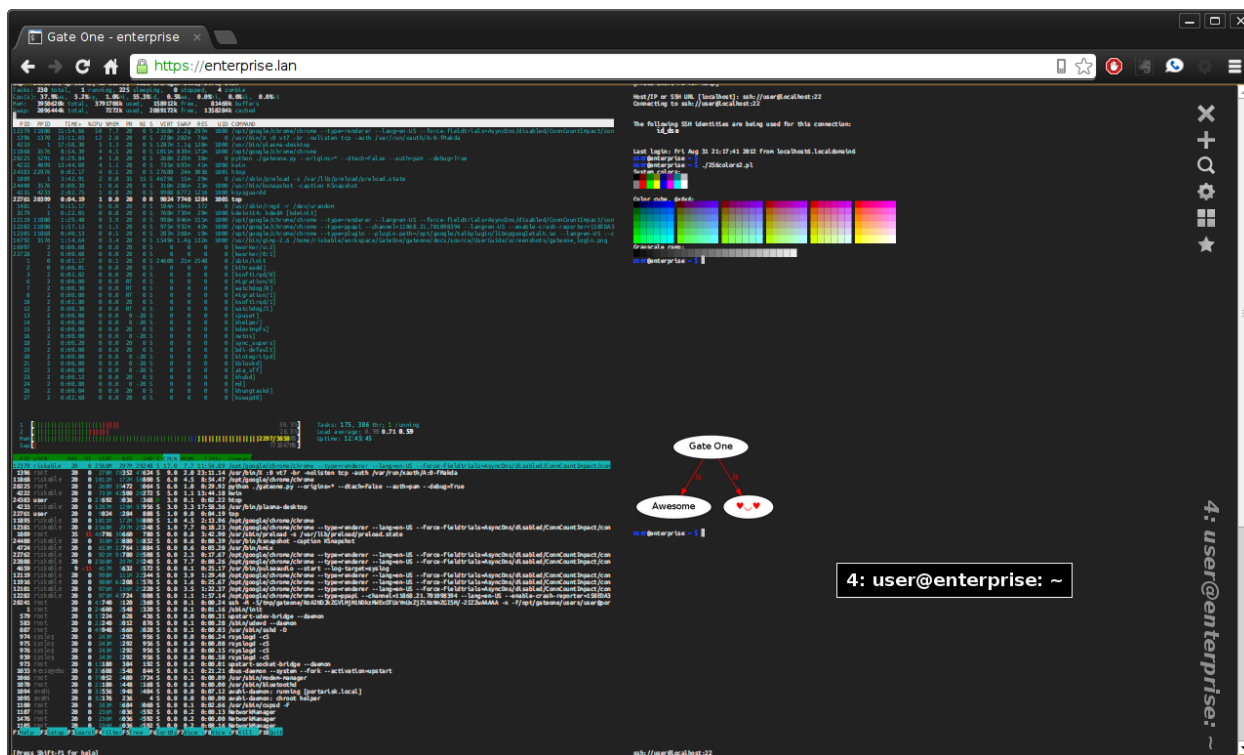


Figure 1.12: The Grid View. The mouse was moved over Terminal 4 in this example, demonstrating the mouseover effect.

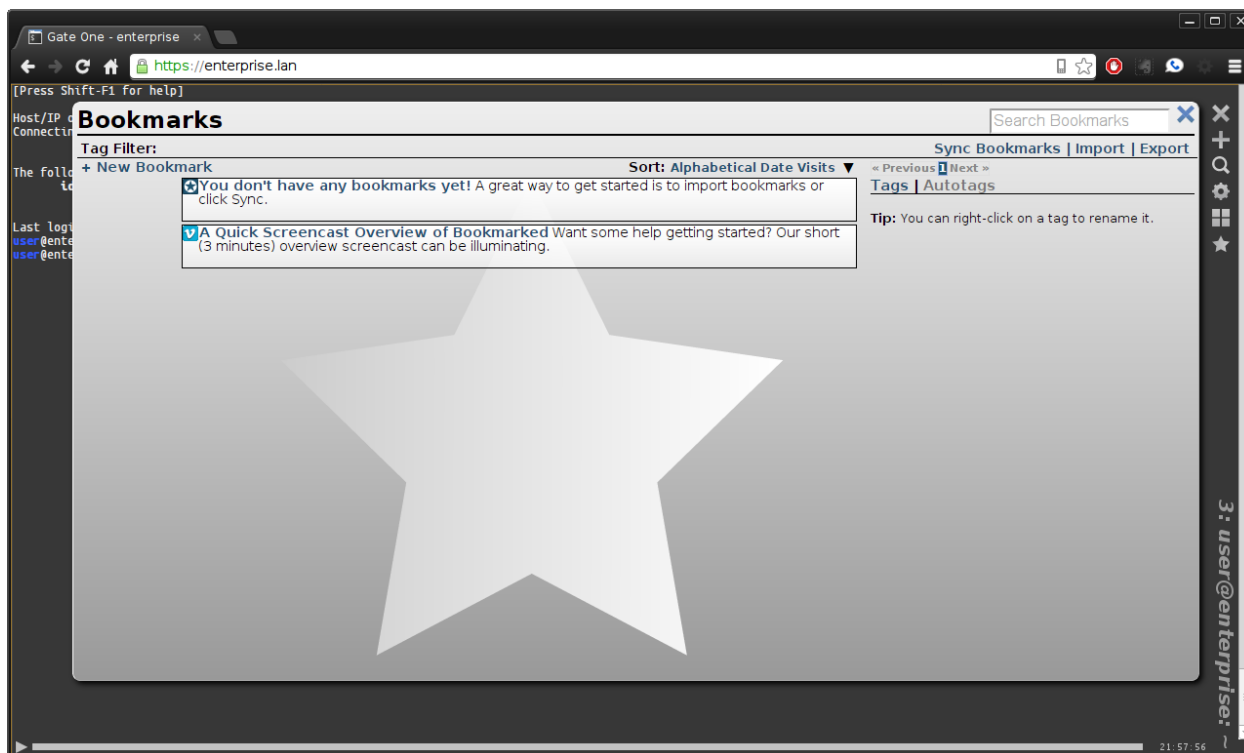


Figure 1.13: No bookmarks yet!

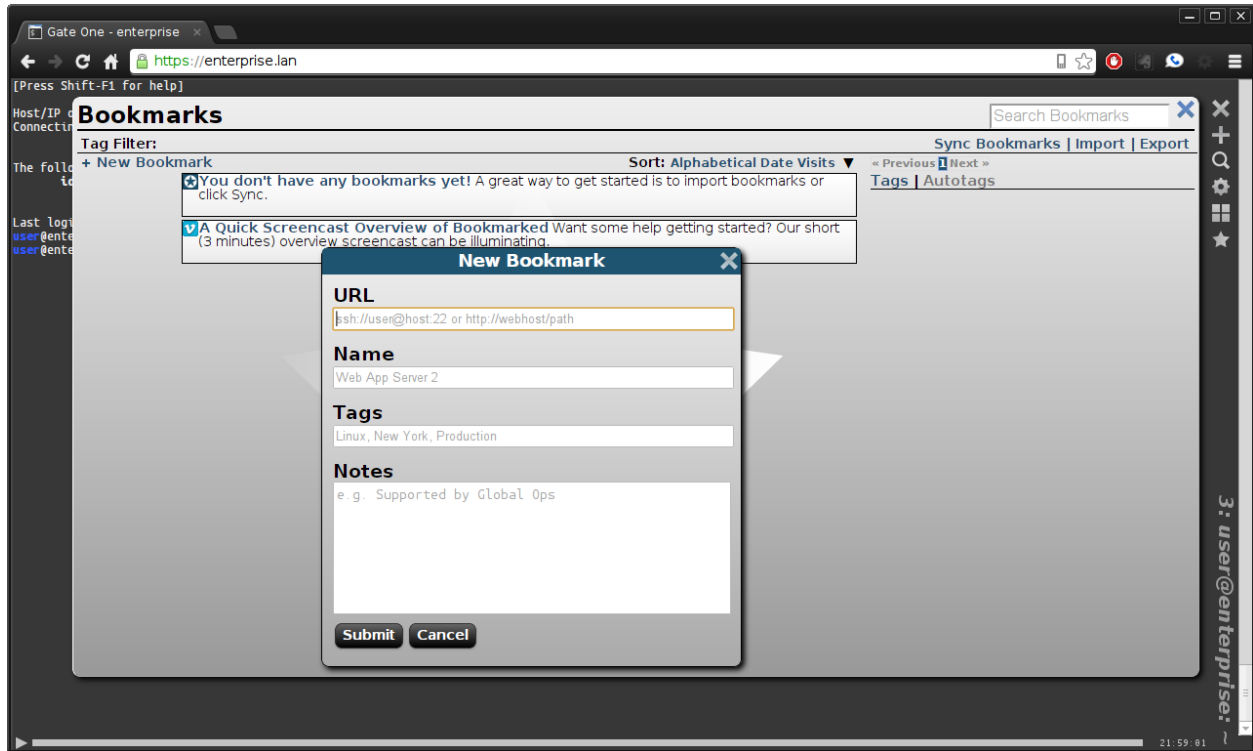


Figure 1.14: The New Bookmark Form.

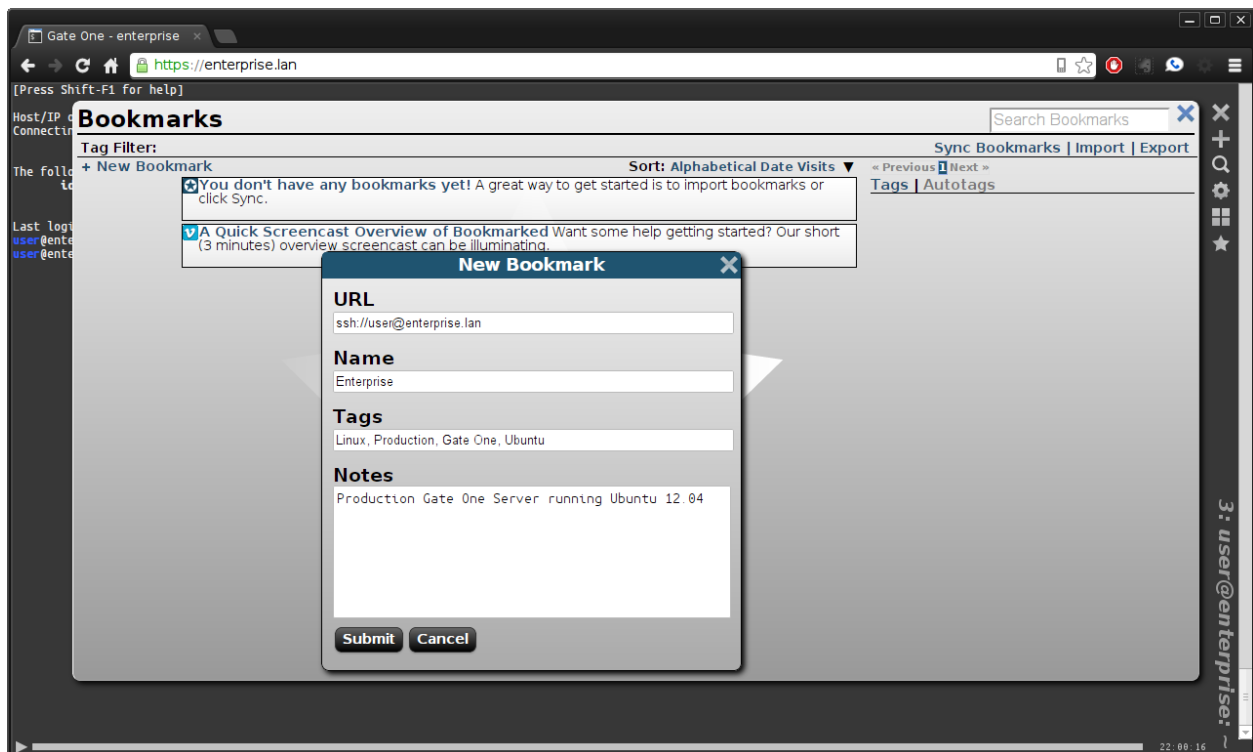


Figure 1.15: SSH Bookmark to the Gate One Demo Server

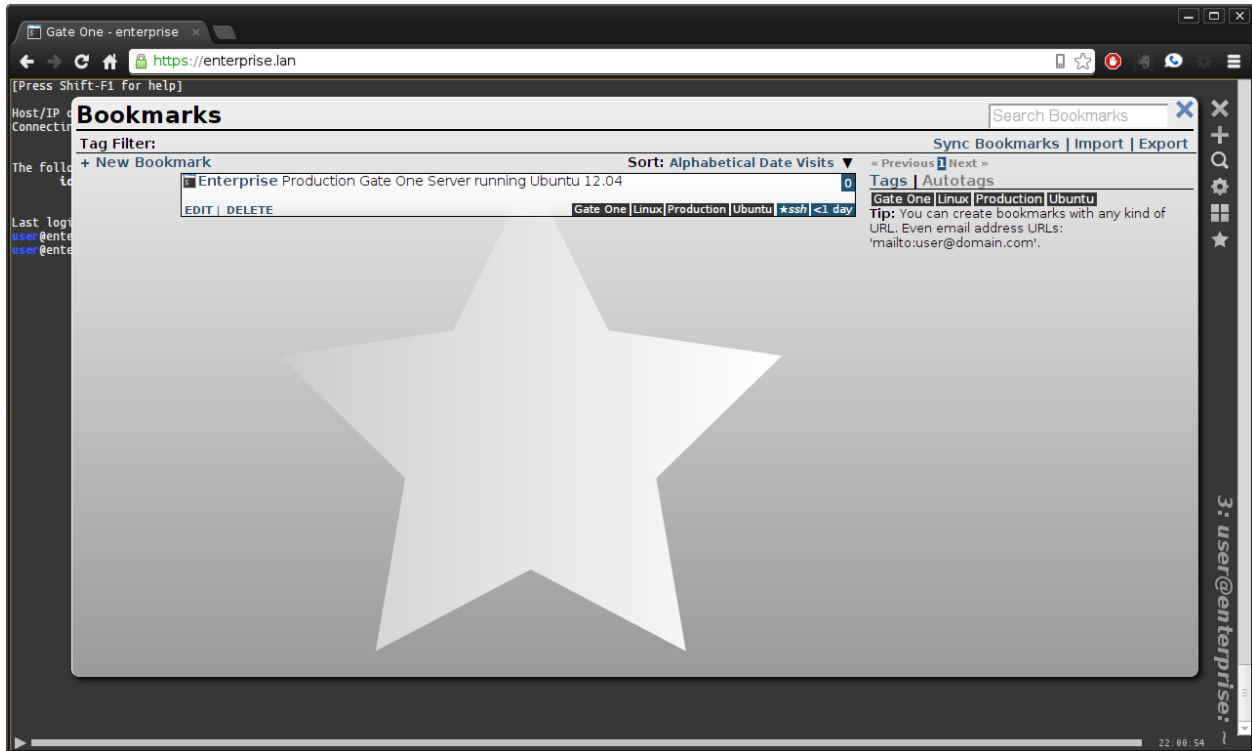


Figure 1.16: Finally, a bookmark!

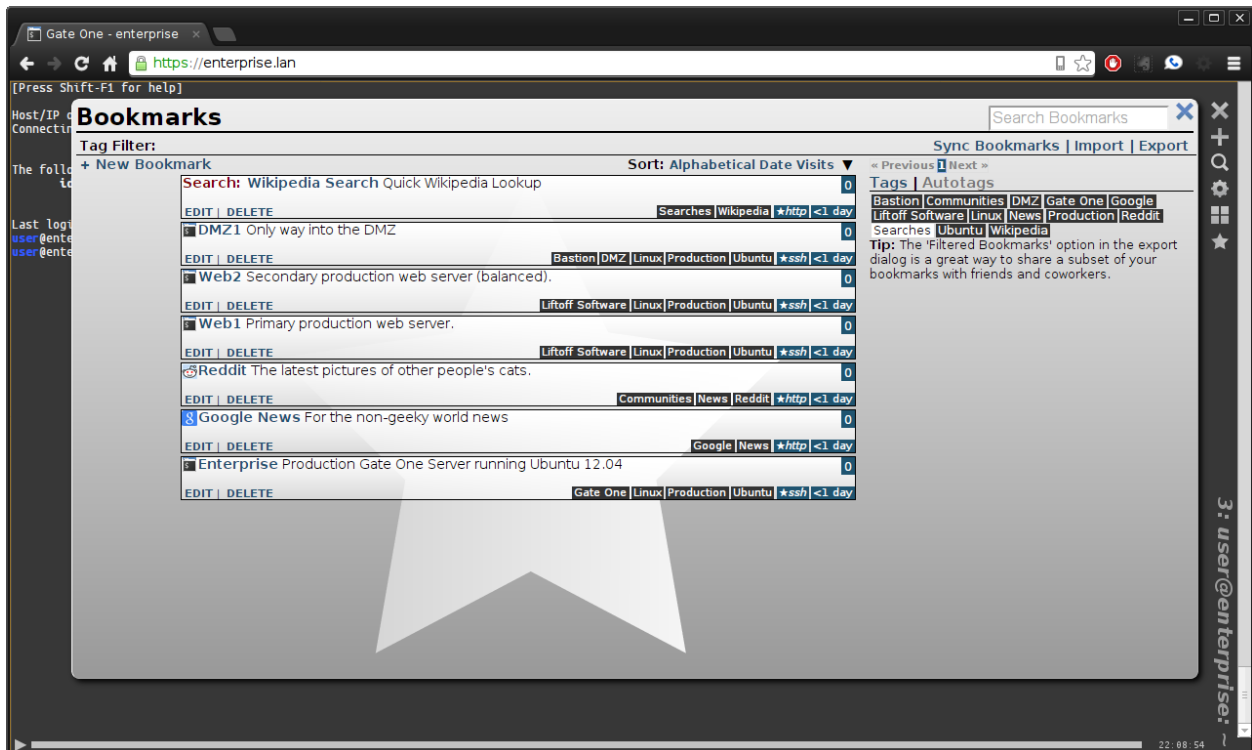


Figure 1.17: In this example we have both SSH bookmarks and an HTTP bookmark.

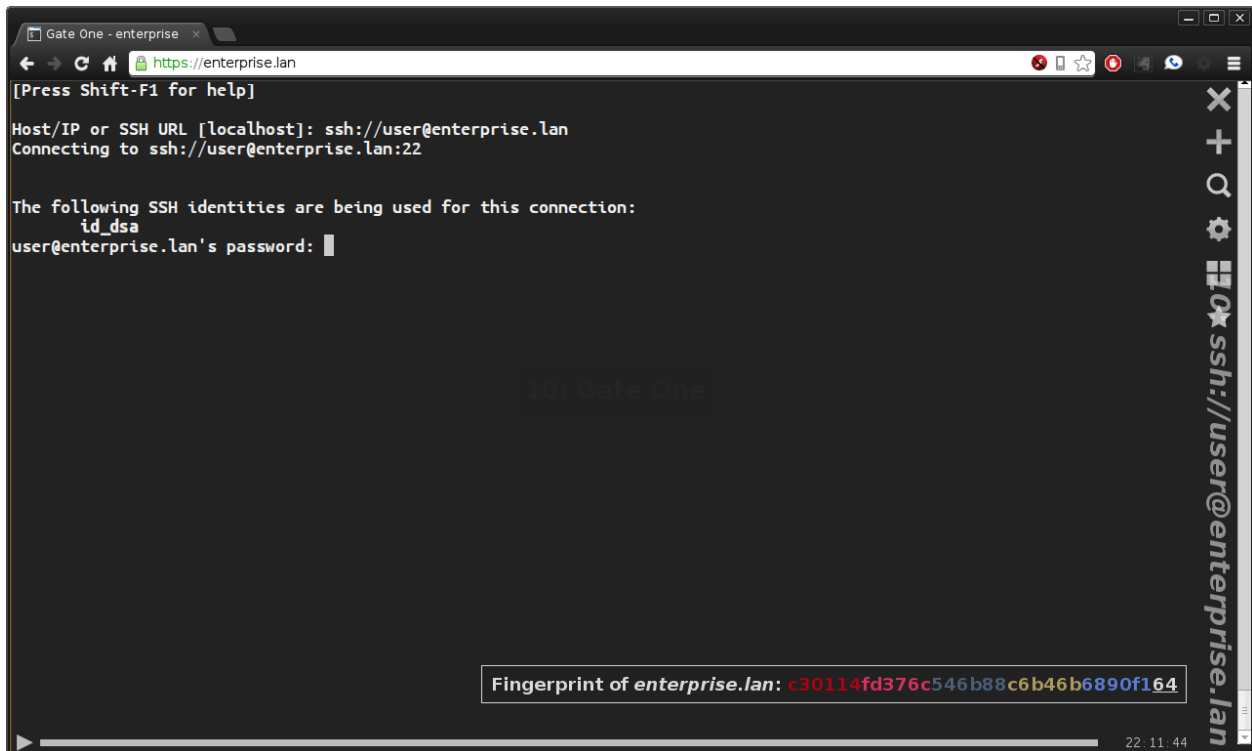


Figure 1.18: This bookmark was automatically opened in a new terminal.

`app_terminal.kill_session(session, kill_dtach=False)`

Terminates all the terminal processes associated with *session*. If *kill_dtach* is True, the dtach processes associated with the session will also be killed.

Note: This function gets appended to the `SESSIONS[session]["kill_session_callbacks"]` list inside of `TerminalApplication.authenticate()`.

`app_terminal.timeout_session(session)`

Attached to Gate One's 'timeout_callbacks'; kills the given session.

If 'dtach' support is enabled the dtach processes associated with the session will also be killed.

class `app_terminal.SharedTermHandler(application, request, **kwargs)`

Renders `shared.html` which allows an anonymous user to view a shared terminal.

class `app_terminal.TermStaticFiles(application, request, **kwargs)`

Serves static files in the `gateone/applications/terminal/static` directory.

Note: This is configured via the `web_handlers` global (a feature inherent to Gate One applications).

class `app_terminal.TerminalApplication(ws)`

A Gate One Application (`G0Application`) that handles creating and controlling terminal applications running on the Gate One server.

initialize()

Called when the WebSocket is instantiated, sets up our WebSocket actions, security policies, and attaches all of our plugin hooks/events.

open()

This gets called at the end of `ApplicationWebSocket.open()` when the `WebSocket` is opened.

send_client_files()

Sends the client our standard CSS and JS.

authenticate()

This gets called immediately after the user is authenticated successfully at the end of `ApplicationWebSocket.authenticate()`. Sends all plugin JavaScript files to the client and triggers the 'terminal:authenticate' event.

on_close()

Removes all attached callbacks and triggers the `terminal:on_close` event.

enumerate_fonts()

Returns a JSON-encoded object containing the installed fonts.

enumerate_colors()

Returns a JSON-encoded object containing the installed text color schemes.

save_term_settings(*term*, *settings*)

Saves whatever *settings* (must be JSON-encodable) are provided in the user's session directory; associated with the given *term*.

The `restore_term_settings` function can be used to restore the provided settings.

Note: This method is primarily to aid dtach support.

restore_term_settings(*term*)

Reads the settings associated with the given *term* that are stored in the user's session directory and applies them to `self.loc_terms[term]`

clear_term_settings(*term*)

Removes any settings associated with the given *term* in the user's `term_settings.json` file (in their session directory).

term_ended(*term*)

Sends the 'term_ended' message to the client letting it know that the given *term* is no more.

add_terminal_callbacks(*term*, *multiplex*, *callback_id*)

Sets up all the callbacks associated with the given *term*, *multiplex* instance and *callback_id*.

remove_terminal_callbacks(*multiplex*, *callback_id*)

Removes all the `Multiplex` and terminal emulator callbacks attached to the given *multiplex* instance and *callback_id*.

new_multiplex(*cmd*, *term_id*, *logging=True*, *encoding='utf-8'*, *debug=False*)

Returns a new instance of `termio.Multiplex` with the proper global and client-specific settings.

cmd The command to execute inside of `Multiplex`.

term_id The terminal to associate with this `Multiplex` or a descriptive identifier (it's only used for logging purposes).

logging If `False`, logging will be disabled for this instance of `Multiplex` (even if it would otherwise be enabled).

encoding The default encoding that will be used when reading or writing to the `Multiplex` instance.

debug If `True`, will enable debugging on the created `Multiplex` instance.

highest_term_num(*location=None*)

Returns the highest terminal number at the given *location* (so we can figure out what's next). If *location* is omitted, uses `self.ws.location`.

send_term_encoding(*term, encoding*)

Sends a message to the client indicating the *encoding* of *term* (in the event that a terminal is reattached or when sharing a terminal).

send_term_keyboard_mode(*term, mode*)

Sends a message to the client indicating the *mode* of *term* (in the event that a terminal is reattached or when sharing a terminal).

set_terminal(*term*)

Sets `self.current_term = *term*` so we can determine where to send keystrokes.

reset_client_terminal(*term*)

Tells the client to reset the terminal (clear the screen and remove scrollbar).

set_title(*term, force=False, save=True*)

Sends a message to the client telling it to set the window title of *term* to whatever comes out of:

```
self.loc_terms[term]['multiplex'].term.get_title() # Whew! Say that three times fast!
```

Example message:

```
{'set_title': {'term': 1, 'title': "user@host"}}
```

If *force* resolves to `True` the title will be sent to the client even if it matches the previously-set title.

if *save* is `True` (the default) the title will be saved via the `TerminalApplication.save_term_settings` function so that it may be restored later (in the event of a server restart—if you've got `dtach` support enabled).

Note: Why the complexity on something as simple as setting the title? Many prompts set the title. This means we'd be sending a 'title' message to the client with nearly every screen update which is a pointless waste of bandwidth if the title hasn't changed.

bell(*term*)

Sends a message to the client indicating that a bell was encountered in the given terminal (*term*). Example message:

```
{'bell': {'term': 1}}
```

mode_handler(*term, setting, boolean*)

Handles mode settings that require an action on the client by passing it a message like:

```
{
    'terminal:set_mode': {
        'mode': setting,
        'bool': True,
        'term': term
    }
}
```

dsr(*term, response*)

Handles Device Status Report (DSR) calls from the underlying program that get caught by the terminal emulator. *response* is what the terminal emulator returns from the `CALLBACK_DSR` callback.

Note: This also handles the CSI DSR sequence.

_send_refresh(*term*, *full=False*)

Sends a screen update to the client.

refresh_screen(*term*, *full=False*, *stream=None*)

Writes the state of the given terminal's screen and scrollbar buffer to the client using `_send_refresh()`. Also ensures that screen updates don't get sent too fast to the client by instituting a rate limiter that also forces a refresh every 150ms. This keeps things smooth on the client side and also reduces the bandwidth used by the application (CPU too).

If *full*, send the whole screen (not just the difference).

full_refresh(*term*)

Calls `self.refresh_screen(*term*, full=True)`

opt_esc_handler(*term*, *multiplex*, *chars*)

Calls whatever function is attached to the 'terminal:opt_esc_handler:<name>' event; passing it the *text* (second item in the tuple) that is returned by `utils.process_opt_esc_sequence()`. Such functions are usually attached via the 'Escape' plugin hook but may also be registered via the usual event method, `:meth'self.on'`:

```
self.on('terminal:opt_esc_handler:somename', some_function)
```

The above example would result in `some_function()` being called whenever a matching optional escape sequence handler is encountered. For example:

```
$ echo -e "\033[_;somename|Text passed to some_function()]\007"
```

Which would result in `some_function()` being called like so:

```
some_function(  
    self, "Text passed to some_function()", term, multiplex)
```

In the above example, *term* will be the terminal number that emitted the event and *multiplex* will be the `termio.Multiplex` instance that controls the terminal.

get_bell()

Sends the bell sound data to the client in the form of a `data::URI`.

get_webworker()

Sends the text of our `term_ww.js` to the client in order to get around the limitations of loading remote Web Worker URLs (for embedding Gate One into other apps).

get_colors(*settings*)

Sends the text color stylesheet matching the properties specified in *settings* to the client. *settings* must contain the following:

colors The name of the CSS text color scheme to be retrieved.

remove_viewer(*term*, *upn=None*)

Disconnects all callbacks attached to the given *term* for the given *upn* and notifies that user that the terminal is no longer shared (so it can be shown to be disconnected at the client).

If *upn* is `None` all users (broadcast viewers included) will have the given *term* disconnected.

notify_permissions()

Sends clients the list of shared terminals if they have been granted access to any shared terminal.

Note: Normally this only gets called from permissions after something changed.

`_shared_terminals_dict(user=None)`

Returns a dict containing information about all shared terminals that the given *user* has access to. If no *user* is given *self.current_user* will be used.

render_256_colors()

Renders the CSS for 256 color support and saves the result as `'256_colors.css'` in Gate One's configured `cache_dir`. If that file already exists and has not been modified since the last time it was generated rendering will be skipped.

Returns the path to that file as a string.

send_256_colors()

Sends the client the CSS to handle 256 color support.

send_print_stylesheet()

Sends the 'templates/printing/printing.css' stylesheet to the client using `ApplicationWebSocket.ws.send_css` with the "media" set to "print".

```
app_terminal.apply_cli_overrides(settings)
```

Updates *settings* in-place with values given on the command line and updates the options *global* with the values from *settings* if not provided on the command line.

```
app_terminal.init(settings)
```

Checks to make sure 50terminal.conf is created if terminal-specific settings are not found in the settings directory.

Also checks to make sure that the logviewer.py script is executable.

Terminal Configuration

Settings The Terminal application stores its settings by default in 'gateone/settings/50terminal.conf'. This file uses Gate One's standard JSON format and should look something like this:

```
// This is Gate One's Terminal application settings file.
```

```
{
  // "*" means "apply to all users" or "default"
  "*": {
    "terminal": { // These settings apply to the "terminal" application
      "commands": {
        "SSH": "/opt/gateone/applications/terminal/plugins/ssh/scripts/ssh_connect.py --logo -S '%SESSION_DIR%/%SESSION_NAME%'",
      },
      "default_command": "SSH",
      "dtach": true,
      "session_logging": true,
      "session_logs_max_age": "30d",
      "syslog_session_logging": false,
      "max_terms": 100
    }
  }
}
```

Tip: If you want Gate One to emulate the system's console (great in the event that SSH is unavailable) you can add "setsid /bin/login" to your commands:

```
"commands": {
  "SSH": "/opt/gateone/applications/terminal/plugins/ssh/scripts/ssh_connect.py --logo -S '%SESSION_DIR%/%SESSION%/%SSH'"
}
```

```
"login": "setsid /bin/login"
}
```

That will allow users to login to the same server hosting Gate One (i.e. just like SSH to localhost). You can set "default_command" to "login" or users can visit https://your-gateone-server/?terminal_cmd=login and all new terminals will be opened using that command.

woff_info.py - Python WOFF Font Inspector Provides a number of functions that can be used to extract the 'name' data from .woff (web font) files.

Note: In most cases .woff files have the metadata stripped (to save space) which is why this module only grabs the 'name' records from the snft (font data) tables.

Example:

```
>>> from pprint import pprint
>>> from woff_info import woff_name_data
>>> woff_path = '/opt/gateone/applications/terminal/static/fonts/ubuntu-mono-normal.woff'
>>> pprint(woff_info(woff_path))
{'Compatible Full': 'Ubuntu Mono',
 'Copyright': 'Copyright 2011 Canonical Ltd. Licensed under the Ubuntu Font Licence 1.0',
 'Designer': 'Dalton Maag Ltd',
 'Designer URL': 'http://www.daltonmaag.com/',
 'Font Family': 'Ubuntu Mono',
 'Font Subfamily': 'Regular',
 'Full Name': 'Ubuntu Mono',
 'ID': 'Ubuntu Mono Regular Version 0.80',
 'Manufacturer': 'Dalton Maag Ltd',
 'Postscript Name': 'UbuntuMono-Regular',
 'Preferred Family': 'Ubuntu Mono',
 'Preferred Subfamily': 'Regular',
 'Trademark': 'Ubuntu and Canonical are registered trademarks of Canonical Ltd.',
 'Vendor URL': 'http://www.daltonmaag.com/',
 'Version': 'Version 0.80'}
```

This script can also be executed on the command line to display the name information for any given WOFF file:

```
user@modern-host:~ $ ./woff_info static/fonts/ubuntu-mono-normal.woff
{
  "Compatible Full": "Ubuntu Mono",
  "Copyright": "Copyright 2011 Canonical Ltd. Licensed under the Ubuntu Font Licence 1.0",
  "Designer": "Dalton Maag Ltd",
  "Designer URL": "http://www.daltonmaag.com/",
  "Font Family": "Ubuntu Mono",
  "Font Subfamily": "Regular",
  "Full Name": "Ubuntu Mono",
  "ID": "Ubuntu Mono Regular Version 0.80",
  "Manufacturer": "Dalton Maag Ltd",
  "Postscript Name": "UbuntuMono-Regular",
  "Preferred Family": "Ubuntu Mono",
  "Preferred Subfamily": "Regular",
  "Trademark": "Ubuntu and Canonical are registered trademarks of Canonical Ltd.",
  "Vendor URL": "http://www.daltonmaag.com/",
  "Version": "Version 0.80"
}
```


..note:

The command line output is JSON so it can be easily used by other programs.

exception `woff_info.BadWoff`

Raised when the name data cannot be extracted from a .woff file (for whatever reason).

`woff_info.woff_name_data(path)`

Returns the 'name' table data from the .woff font file at the given *path*.

Note: Only returns the English language stuff.

JavaScript Code

terminal.js Gate One's bundled Terminal application JavaScript ([source](#)).

`GateOne.Terminal.__new__(settings[, where])`

Called when a user clicks on the Terminal Application in the New Workspace Workspace (or anything that happens to call `__new__()`).

Settings An object containing the settings that will control how the new terminal is created. Typically contains the application's 'info' data from the server.

Where An optional querySelector-like string or DOM node where the new Terminal should be placed. If not given a new workspace will be created to contain the Terminal.

`GateOne.Terminal.hasVoiceExt()`

This function returns true if an extension is detected that converts phone numbers into clickable links (e.g. Google Voice). Such browser extensions can have a *severe* negative performance impact while using terminals (every screen update requires a re-scan of the entire page!).

`GateOne.Terminal.dbReady(db)`

Sets `GateOne.Terminal.dbReady = true` so that we know when the 'terminal' database is available & ready (indexedDB databases are async).

`GateOne.Terminal.createPrefsPanel()`

Creates the terminal preferences and adds them to the global preferences panel.

`GateOne.Terminal.enumerateCommandsAction(messageObj)`

Attached to the 'terminal:commands_list' WebSocket action; stores `messageObj['commands']` in `GateOne.Terminal.commandsList`.

`GateOne.Terminal.enumerateFontsAction(messageObj)`

Attached to the 'terminal:fonts_list' WebSocket action; updates the preferences panel with the list of fonts stored on the server and stores the list in `GateOne.Terminal.fontsList`.

`GateOne.Terminal.enumerateColorsAction(messageObj)`

Attached to the 'terminal:colors_list' WebSocket action; updates the preferences panel with the list of text color schemes stored on the server.

`GateOne.Terminal.onResizeEvent()`

Attached to the `go:update_dimensions` event; calls `GateOne.Terminal.sendDimensions()` for all terminals to ensure the new dimensions get applied.

`GateOne.Terminal.reconnectEvent()`

Attached to the `go:connection_established` event; closes all open terminals so that `GateOne.Terminal.reattachTerminalsAction()` can do its thing.

`GateOne.Terminal.connectionError()`

This function gets attached to the “go:connection_error” event; closes all open terminals.

`GateOne.Terminal.getOpenTerminals()`

Requests a list of open terminals on the server via the ‘terminal:get_terminals’ WebSocket action. The server will respond with a ‘terminal:terminals’ WebSocket action message which calls `GateOne.Terminal.reattachTerminalsAction()`.

`GateOne.Terminal.sendString(chars[, term])`

Like `sendChars()` but for programmatic use. *chars* will be sent to *term* on the server.

If *term* is not given the currently-selected terminal will be used.

`GateOne.Terminal.killTerminal(term)`

Tells the server got close the given *term* and kill the underlying process.

`GateOne.Terminal.refresh(term)`

Tells the Gate One server to send a screen refresh (using the diff method).

Note: This function is only here for debugging purposes. Under normal circumstances difference-based screen refreshes are initiated at the server.

`GateOne.Terminal.fullRefresh(term)`

Tells the Gate One server to send a full screen refresh to the client for the given *term*.

`GateOne.Terminal.loadFont(font[, size])`

Tells the server to perform a sync of the given *font* with the client. If *font* is not given, will load the font set in `GateOne.prefs.font`.

Optionally, a *size* may be chosen. It must be a valid CSS ‘font-size’ value such as ‘0.9em’, ‘90%’, ‘12pt’, etc.

`GateOne.Terminal.loadTextColors(colors)`

Tells the server to perform a sync of the given *colors* (terminal text colors) with the client. If *colors* is not given, will load the colors set in `GateOne.prefs.colors`.

`GateOne.Terminal.displayTermInfo(term)`

Displays the given term’s information as a psuedo tooltip that eventually fades away.

`GateOne.Terminal.sendDimensions([term[, ctrl_l]])`

Detects and sends the given term’s dimensions (rows/columns) to the server.

If no *term* is given it will send the dimensions of the currently-selected terminal to the server which will be applied to all terminals.

`GateOne.Terminal.setTitle(term, text)`

Sets the visible title of *term* to *text* appropriately based on whether or not this is a popup terminal or just a regular one.

Note: This function does *not* set the ‘title’ or ‘X11Title’ attributes of the `GateOne.Terminal.terminals[term]` object. That is handled by `GateOne.Terminal.setTitleAction()`.

`GateOne.Terminal.setTitleAction(titleObj)`

Sets the title of *titleObj.term* to *titleObj.title*.

`GateOne.Terminal.resizeAction(message)`

Called when the server sends the `terminal:resize` WebSocket action. Sets the ‘rows’ and ‘columns’ values inside `GateOne.Terminal.terminals[message.term]` and sets the same values inside the Info & Tools panel.

GateOne.Terminal.**paste**(*e*)

This gets attached to Shift-Insert (KEY_INSERT) as a shortcut in order to support pasting.

GateOne.Terminal.**writeScrollbar**(*term*, *scrollback*)

Writes the given *scrollback* buffer for the given *term* to localStorage.

GateOne.Terminal.**applyScreen**(*screen*[, *term*[, *noUpdate*]])

Uses *screen* (array of HTML-formatted lines) to update *term*.

If *term* is omitted localStorage[prefix+selectedTerminal] will be used.

If *noUpdate* is true the array that holds the current screen in GateOne.Terminal.terminals will not be updated (useful for temporary screen replacements).

Note: Lines in *screen* that are empty strings or null will be ignored (so it is safe to pass a full array with only a single updated line).

GateOne.Terminal.**alignTerminal**(*term*)

Uses a CSS3 transform to move the terminal <pre> element upwards just a bit so that the scrollbar buffer isn't visible unless you actually scroll. This improves the terminal's overall appearance considerably because the bottoms of characters in the scrollbar buffer tend to look like graphical glitches.

GateOne.Terminal.**termUpdateFromWorker**(*e*)

This function gets assigned to the termUpdatesWorker.onmessage() event; Whenever the Web Worker completes processing of the incoming screen it posts the result to this function. It takes care of updating the terminal on the page, storing the scrollbar buffer, and finally triggers the "terminal:term_updated" event passing the terminal number as the only argument.

GateOne.Terminal.**loadWebWorkerAction**(*source*)

Loads our Web Worker given it's *source* (which is sent to us over the WebSocket which is a clever workaround to the origin limitations of Web Workers =).

GateOne.Terminal.**updateTerminalAction**(*termUpdateObj*)

Takes the updated screen information from *termUpdateObj* and posts it to the 'term_ww.js' Web Worker to be processed.

Note: The Web Worker is important because it allows offloading of CPU-intensive tasks like linkification and text transforms so they don't block screen updates

GateOne.Terminal.**notifyInactivity**(*term*)

Notifies the user of inactivity in *term*.

GateOne.Terminal.**notifyActivity**(*term*)

Notifies the user of activity in *term*.

GateOne.Terminal.**newPastearea**()

Returns a 'pastearea' (textarea) element meant for placement above terminals for the purpose of enabling proper copy & paste.

GateOne.Terminal.**newTerminal**([*term*[, *settings*[, *where*]]])

Adds a new terminal to the grid and starts updates with the server.

If *term* is provided, the created terminal will use that number.

If *settings* (associative array) are provided the given parameters will be applied to the created terminal's parameters in GateOne.Terminal.terminals[term] as well as sent as part of the 'terminal:new_terminal' WebSocket action. This mechanism can be used to spawn terminals using different 'commands' that have been configured on the server. For example:

```
> // Creates a new terminal that spawns whatever command is set as 'login' in Gate One's settings:
> GateOne.Terminal.newTerminal(null, {'command': 'login'});
```

If *where* is provided, the new terminal element will be appended like so: `where.appendChild(<new terminal element>);` Otherwise the terminal will be added to the grid.

Terminal types are sent from the server via the 'terminal_types' action which sets up `GateOne.terminalTypes`. This variable is an associative array in the form of: `{'term type': {'description': 'Description of terminal type', 'default': true/false, <other, yet-to-be-determined metadata>}}`.

```
GateOne.Terminal.closeTerminal(term[, noCleanup[, message[, sendKill]]])
```

Arguments

- **term** (*number*) – The terminal to close.
- **noCleanup** (*boolean*) – If true the terminal's metadata in `localStorage/IndexedDB` (i.e. scrollback buffer) will not be removed.
- **message** (*string*) – An optional message to display to the user after the terminal is close.
- **sendKill** (*boolean*) – If undefined or true, will tell the server to kill the process associated with the given *term* (i.e. close it for real).

Closes the given terminal (*term*) and tells the server to end its running process.

```
GateOne.Terminal.popupTerm([term])
```

Opens a dialog with a terminal contained within. If *term* is given the created terminal will use that number.

The *options* argument may contain the following:

global If true the dialog will be appended to `GateOne.node` (e.g. `#gateone`) instead of the current workspace.

where If provided the popup terminal will be placed within the given element.

If the terminal inside the dialog ends it will be closed automatically. If the user closes the dialog the terminal will be closed automatically as well.

```
GateOne.Terminal.setTerminal(term)
```

Sets the 'selectedTerminal' value in `localStorage` and sends the 'terminal:set_terminal' WebSocket action to the server to let it know which terminal is currently active.

This function triggers the 'terminal:set_terminal' event passing the terminal number as the only argument.

```
GateOne.Terminal.switchTerminal(term)
```

Calls `GateOne.Terminal.setTerminal(*term*)` then triggers the 'terminal:switch_terminal' event passing *term* as the only argument.

```
GateOne.Terminal.setActive([term])
```

Removes the 'inactive' class from the given *term*.

If *term* is not given the currently-selected terminal will be used.

```
GateOne.Terminal.isActive()
```

Returns true if a terminal is the current application (selected by the user)

```
GateOne.Terminal.switchTerminalEvent(term)
```

This gets attached to the 'terminal:switch_terminal' event in `GateOne.Terminal.init()`; performs a number of actions whenever the user changes the current terminal.

`GateOne.Terminal.switchWorkspaceEvent(workspace)`

Called whenever Gate One switches to a new workspace; checks whether or not this workspace is home to a terminal and calls `switchTerminalEvent()` on said terminal (to make sure input is enabled and it is scrolled to the bottom).

`GateOne.Terminal.workspaceClosedEvent(workspace)`

Attached to the `go:close_workspace` event; closes any terminals that are attached to the given *workspace*.

`GateOne.Terminal.swappedWorkspacesEvent(ws1, ws2)`

Attached to the `go:swapped_workspaces` event; updates `GateOne.Terminal.terminals` with the correct workspace attributes if either contains terminals.

`GateOne.Terminal.printScreen(term)`

Prints *just* the screen (no scrollbar) of the given *term*. If *term* is not provided the currently-selected terminal will be used.

`GateOne.Terminal.hideIcons()`

Hides the Terminal's toolbar icons (i.e. when another application is running).

`GateOne.Terminal.showIcons()`

Shows (unhides) the Terminal's toolbar icons (i.e. when another application is running).

`GateOne.Terminal.bellAction(bellObj)`

Attached to the 'terminal:bell' WebSocket action; plays a bell sound and pops up a message indicating which terminal issued a bell.

`GateOne.Terminal.playBell()`

Plays the bell sound without any visual notification.

`GateOne.Terminal.updateDimensions()`

This gets attached to the "go:update_dimensions" event which gets called whenever the window is resized. It makes sure that all the terminal containers are of the correct dimensions.

`GateOne.Terminal.enableScrollback([term])`

Replaces the contents of the selected/active terminal scrollbar buffer with the complete latest scrollbar buffer from `GateOne.Terminal.terminals[*term*]`.

If *term* is given, only enable scrollbar for that terminal.

`GateOne.Terminal.toggleScrollback()`

Enables or disables the scrollbar buffer (to hide or show the scrollbars).

`GateOne.Terminal.clearScrollback(term)`

Empties the scrollbar buffer for the given *term* in memory, in localStorage, and in the DOM.

`GateOne.Terminal.clearScreen(term)`

Clears the screen of the given *term* in memory and in the DOM.

Note: The next incoming screen update from the server will likely re-populate most if not all of the screen.

`GateOne.Terminal.getLocations()`

Sends the `terminal:get_locations` WebSocket action to the server. This will ultimately trigger the `GateOne.Terminal.locationsAction()` function.

`GateOne.Terminal.locationsAction(locations)`

Attached to the `terminal:term_locations` WebSocket action, triggers the `terminal:term_locations` event (in case someone wants to do something with that information).

GateOne.Terminal.**relocateWorkspaceEvent**(*workspace*, *location*)

Attached to the go:relocate_workspace event; calls GateOne.Terminal.relocateTerminal() if the given *workspace* has a terminal contained within it.

GateOne.Terminal.**relocateTerminal**(*term*, *location*)

Arguments

- **term** (*number*) – The number of the terminal to move (e.g. 1).
- **location** (*string*) – The 'location' where the terminal will be moved (e.g. 'window2').

Moves the given *term* to the given *location* (aka window) by sending the appropriate message to the Gate One server.

GateOne.Terminal.**changeLocation**(*location*[, *settings*])

Attached to the go:set_location event, removes all terminals from the current view and opens up all the terminals at the new *location*. If there are currently no terminals at *location* a new terminal will be opened automatically.

To neglect opening a new terminal automatically provide a settings object like so:

```
>>> GateOne.Terminal.changeLocation('window2', {'new_term': false});
```

GateOne.Terminal.**reconnectTerminalAction**(*message*)

Called when the server reports that the terminal number supplied via the terminal:new_terminal WebSocket action already exists.

This method also gets called when a terminal is moved from one 'location' to another.

GateOne.Terminal.**moveTerminalAction**(*obj*)

Attached to the terminal:term_moved WebSocket Action, closes the given *term* with a slightly different message than closeTerminal().

GateOne.Terminal.**reattachTerminalsAction**(*terminals*)

Called after we authenticate to the server, this function is attached to the terminal:terminals WebSocket action which is the server's way of notifying the client that there are existing terminals.

If we're reconnecting to an existing session, running terminals will be recreated/reattached.

If this is a new session (and we're not in embedded mode), a new terminal will be created.

GateOne.Terminal.**recordPanePositions**()

Records the position of each terminal in each GateOne.Visual.Pane() that exists on the page (so they can be resumed properly).

GateOne.Terminal.**resumePanePosition**(*term*)

Uses GateOne.Terminal.terminals[term].metadata.panePosition to create a new terminal in that exact spot. New :js:meth'GateOne.Visual.Pane' objects will be created as necessary to ensure the terminal winds up where it was previously.

GateOne.Terminal.**setModeAction**(*modeObj*)

Set the given terminal mode (e.g. application cursor mode aka appmode). *modeObj* is expected to be something like this:

```
{'mode': '1', 'term': '1', 'bool': true}
```

GateOne.Terminal.**registerTextTransform**(*name*, *pattern*, *newString*)

Adds a new or replaces an existing text transformation to GateOne.Terminal.textTransforms using *pattern* and *newString* with the given *name*. Example:

```
var pattern = /(\\bIM\\d{9,10}\\b)/g,  
    newString = "<a href='https://support.company.com/tracker?ticket=$1' target='new'>$1</a>";  
GateOne.Terminal.registerTextTransform("ticketIDs", pattern, newString);
```

Would linkify text matching that pattern in the terminal.

For example, if you typed "Ticket number: IM123456789" into a terminal it would be transformed thusly:

```
"Ticket number: <a href='https://support.company.com/tracker?ticket=IM123456789' target='new'>IM123456789</a>"
```

Alternatively, a function may be provided instead of *pattern*. In this case, each line will be transformed like so:

```
line = pattern(line);
```

Note: *name* is only used for reference purposes in the textTransforms object (so it can be removed or replaced later).

Tip: To match the beginning of a line use '\\n' instead of '^'. This is necessary because the entire screen is matched at once as opposed to line-by-line.

GateOne.Terminal.unregisterTextTransform(*name*)

Removes the given text transform from GateOne.Terminal.textTransforms.

GateOne.Terminal.resetTerminalAction(*term*)

Clears the screen and the scrollbar buffer of the given *term* (in memory, localStorage, and in the DOM).

GateOne.Terminal.termEncodingAction(*message*)

Handles the 'terminal:encoding' WebSocket action that tells us the encoding that is set for a given terminal. The expected message format:

Arguments

- **message.term** (*string*) – The terminal in question.
- **message['encoding']** (*string*) – The encoding to set on the given terminal.

Note: The encoding value here is only used for informational purposes. No encoding/decoding happens at the client.

GateOne.Terminal.termKeyboardModeAction(*message*)

Handles the 'terminal:keyboard_mode' WebSocket action that tells us the keyboard mode that is set for a given terminal. The expected message format:

Arguments

- **message.term** (*string*) – The terminal in question.
- **message['mode']** (*string*) – The keyboard mode to set on the given terminal. E.g. 'default', 'sco', 'xterm', 'linux', etc

Note: The keyboard mode value is only used by the client. There's no server-side functionality related to keyboard modes other than the fact that it remembers the setting.

GateOne.Terminal.xtermEncode(*number*)

Encodes the given *number* into a single character using xterm's encoding scheme. e.g. to convert

mouse coordinates for use in an escape sequence.

The xterm encoding scheme takes the ASCII value of (*number* + 32). So the number one would be ! (exclamation point), the number two would be " (double-quote), the number three would be # (hash), and so on.

Note: This encoding mechanism has the unfortunate limitation of only being able to encode up to the number 233.

GateOne.Terminal.**highlight**(*text*[, *term*])

Highlights all occurrences of the given *text* inside the given *term* by wrapping it in a span like so:

```
<span class="highlight">text</span>
```

If *term* is not provided the currently-selected terminal will be used.

GateOne.Terminal.**unHighlight**()

Undoes the results of GateOne.Terminal.highlight().

GateOne.Terminal.**highlightTexts**

An object that holds all the words the user wishes to stay persistently highlighted (even after screen updates and whatnot) across all terminals.

Note: Word highlighting that is specific to individual terminals is stored in GateOne.Terminal.terminals[*term*].

GateOne.Terminal.**highlightDialog**()

Opens a dialog where users can add/remove text they would like to be highlighted on a semi-permanent basis (e.g. even after a screen update).

GateOne.Terminal.**showSuspended**()

Displays a little widget in the terminal that indicates that output has been suspended.

GateOne.Terminal.**scrollPageUp**([*term*])

Scrolls the given *term* one page up. If *term* is not given the currently-selected terminal will be used.

GateOne.Terminal.**scrollPageDown**([*term*])

Scrolls the given *term* one page up. If *term* is not given the currently-selected terminal will be used.

GateOne.Terminal.**chooser**()

Pops up a dialog where the user can immediately switch to any terminal in any 'location'.

Note: If the terminal is in a different location the current location will be changed along with all terminals before the switch is made.

GateOne.Terminal.**sharePermissions**(*term*, *permissions*)

Sets the sharing *permissions* of the given *term*. The *permissions* must be an object that contains one or both of the following:

read An array of users that will be given read-only access to the given *term*.

write An array of users that will be given write access to the given *term*.

Note: *read* and *write* may be given as a string if only one user's permissions are being modified.

Example:


```
>>> GateOne.Terminal.sharePermissions(1, {'read': 'AUTHENTICATED', 'write': ["bob@company", "joe@company"]});
```

Note: If a user is granted write permission to a terminal they will automatically be granted read permission.

GateOne.Terminal.**shareDialog**(*term*)

Opens a dialog where the user can share a terminal or modify the permissions on a terminal that is already shared.

GateOne.Terminal.**setShareID**(*term*[, *shareID*])

Sets the share ID of the given *term*. If a *shareID* is not provided one will be generated automatically (by the server).

GateOne.Terminal.**attachSharedTerminal**(*shareID*[, *password*[, *metadata*]])

Opens the terminal associated with the given *shareID*.

If a *password* is given it will be used to attach to the shared terminal.

If *metadata* is given it will be passed to the server to be used for extra logging and providing additional details about who is connecting to shared terminals.

GateOne.Terminal.**detachSharedTerminal**(*term*)

Tells the server that we no longer wish to view the terminal associated with the given *term* (local terminal number).

GateOne.Terminal.**listSharedTerminals**()

Sends the terminal:list_shared_terminals WebSocket action to the server which will reply with the terminal:shared_terminals WebSocket action (if the user is allowed to list shared terminals).

GateOne.Terminal.**sharedTerminalsAction**(*message*)

Attached to the terminal:shared_terminals WebSocket action; stores the list of terminals that have been shared with the user in GateOne.Terminal.sharedTerminals and triggers the terminal:shared_terminals event.

GateOne.Terminal.**sharedTerminalsDialog**()

Opens up a dialog where the user can open terminals that have been shared with them.

GateOne.Terminal.**shareInfo**(*term*)

Displays a dialog that lists the current viewers (along with their permissions) of a given *term*.

GateOne.Terminal.**shareWidget**(*term*)

Adds a terminal sharing widget to the given *term* (number) that provides sharing controls and information (e.g. number of viewers).

GateOne.Terminal.**sharedTermObj**(*term*)

Returns the object matching the given *term* from GateOne.Terminal.sharedTerminals.

GateOne.Terminal.**shareID**(*term*)

Returns the share ID for the given *term* (if any).

GateOne.Terminal.**shareBroadcastURL**(*term*)

Returns the broadcast URL for the given *term* (if any).

GateOne.Terminal.**_trimmedScreen**(*screen*)

Returns *screen* with all trailing empty lines removed.

GateOne.Terminal.**lastLines**(*n*[, *term*])

Returns the last *n* non-blank (trimmed) line in the terminal. Useful for pattern matching.

If *term* is not given the localStorage[prefix+'selectedTerminal'] will be used.

terminal_input.js Gate One's bundled Terminal application JavaScript (Input module) ([source](#)).

GateOne.Terminal.Input

Terminal-specific keyboard and mouse input stuff.

GateOne.Terminal.Input.init()

Creates `GateOne.Terminal.Input.inputNode` to capture keys/IME composition and attaches appropriate events.

GateOne.Terminal.Input.sendChars()

pop()s out the current `charBuffer` and sends it to the server.

GateOne.Terminal.Input.onMouseWheel(e)

Attached to the mousewheel event on the Terminal application container; calls `preventDefault()` if "mouse motion" event tracking mode is enabled and instead sends equivalent xterm escape sequences to the server to emulate mouse scroll events.

If the `Alt` key is held the user will be able to scroll normally.

GateOne.Terminal.Input.onMouseMove(e)

Attached to the contextmenu event on the Terminal application container; calls `preventDefault()` if "mouse motion" event tracking mode is enabled to prevent the usual context menu from popping up.

If the `Alt` key is held while right-clicking the normal context menu will appear.

GateOne.Terminal.Input.onMouseMove(e)

Attached to the mousemove event on the Terminal application container when mouse event tracking is enabled; pre-sets `GateOne.Terminal.mouseUpEscSeq` with the current mouse coordinates.

GateOne.Terminal.Input.onMouseDown(e)

Attached to the mousedown event on the Terminal application container; performs the following actions based on which mouse button was used:

- Left-click: Hides the pastearea.
- Right-click: If no text is highlighted in the terminal, makes sure the pastearea is visible and has focus.
- Middle-click: Makes sure the pastearea is visible and has focus so that middle-click-to-paste events (X11) will work properly. Alternatively, if there is highlighted text in the terminal a paste event will be emulated (regardless of platform).

GateOne.Terminal.Input.onMouseUp(e)

Attached to the mouseup event on the Terminal application container; prevents the pastearea from being shown if text is highlighted in the terminal (so users can right-click to copy). Also prevents the pastearea from being instantly re-enabled when clicking in order to allow double-click events to pass through to the terminal (to highlight words).

The last thing this function does every time it is called is to change focus to `GateOne.Terminal.Input.inputNode`.

GateOne.Terminal.Input.onCopy()

Returns all 'pastearea' elements to a visible state after a copy operation so that the browser's regular context menu will be usable again (for pasting). Also displays a message to the user letting them know that the text was copied successfully (because having your highlighted text suddenly disappear isn't that intuitive).

GateOne.Terminal.Input.capture()

Sets focus on the terminal and attaches all the relevant events (mousedown, mouseup, keydown, etc).

`GateOne.Terminal.Input.disableCapture(e[, force])`

Disables the various input events that capture mouse and keystroke events. This allows things like input elements and forms to work properly (so keystrokes can pass through without intervention).

`GateOne.Terminal.Input.onPaste(e)`

Attached to the 'paste' event on the terminal application container; converts pasted text to plaintext and sends it to the selected terminal.

`GateOne.Terminal.Input.queue(text)`

Adds 'text' to the `GateOne.Terminal.Input.charBuffer` Array (to be sent to the server when ready via `GateOne.Terminal.sendChars()`).

`GateOne.Terminal.Input.queue(cars)`

Prepends the ESC key string (`String.fromCharCode(27)`) to special character sequences (e.g. PgUp, PgDown, Arrow keys, etc) before adding them to the charBuffer

`GateOne.Terminal.Input.onCompositionStart(e)`

Called when we encounter the compositionstart event which indicates the use of an IME. That would most commonly be a mobile phone software keyboard or foreign language input methods (e.g. Anthy for Japanese, Chinese, etc).

Ensures that `GateOne.Terminal.Input.inputNode` is visible and as close to the cursor position as possible.

`GateOne.Terminal.Input.onCompositionEnd(e)`

Called when we encounter the compositionend event which indicates the IME has completed a composition. Sends what was composed to the server and ensures that `GateOne.Terminal.Input.inputNode` is emptied & hidden.

`GateOne.Terminal.Input.onCompositionUpdate(e)`

Called when we encounter the 'compositionupdate' event which indicates incoming characters; sets `GateOne.Terminal.Input.composition`.

`GateOne.Terminal.Input.onKeyUp(e)`

Called when the terminal encounters a keyup event; just ensures that `GateOne.Terminal.Input.inputNode` is emptied so we don't accidentally send characters we shouldn't.

`GateOne.Terminal.Input.onInput(e)`

Attached to the input event on `GateOne.Terminal.Input.inputNode`; sends its contents. If the user is in the middle of composing text via an IME it will wait until their composition is complete before sending the characters.

`GateOne.Terminal.Input.onKeyDown(e)`

Handles keystroke events by determining which kind of event occurred and how/whether it should be sent to the server as specific characters or escape sequences.

`GateOne.Terminal.Input.execKeystroke(e)`

For the Terminal application, executes the keystroke or shortcut associated with the given key-down event (*e*).

`GateOne.Terminal.Input.emulateKey(e, skipF11check)`

This method handles all regular keys registered via onkeydown events (not onkeypress) If *skipF11check* is true, the F11 (fullscreen check) logic will be skipped.

Note: Shift+key also winds up being handled by this function.

`GateOne.Terminal.Input.emulateKeyCombo(e)`

This method translates ctrl/alt/meta key combos such as Ctrl-c into their string equivalents

using `GateOne.Terminal.Input.keyTable` and sends them to the server.

Plugin Code

The Bookmarks Plugin

JavaScript `bookmarks.js` - The client-side portion of Gate One's Bookmarks plugin.

`GateOne.Bookmarks.addFilterDateTag(bookmarks, dateTag)`

Adds the given *dateTag* to the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.addFilterTag(bookmarks, tag)`

Adds the given *tag* to the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.addFilterURLTypeTag(bookmarks, typeTag)`

Adds the given *typeTag* to the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.addTagToBookmark(URL, tag)`

Adds the given *tag* to the bookmark object associated with *URL*.

`GateOne.Bookmarks.allTags()`

Returns an array of all the tags in `localStorage[GateOne.prefs.prefix+'bookmarks']` ordered alphabetically.

`GateOne.Bookmarks.bookmarks`

All the user's bookmarks are stored in this array which is stored/loaded from `localStorage[GateOne.prefs.prefix+'bookmarks']`. Each bookmark consists of the following data structure:

```
{
  created: 1356974567922,
  images: {favicon: "data:image/x-icon;base64,<gobbledygook>"},
  name: "localhost",
  notes: "Login to the Gate One server itself",
  tags: ["Linux", "Ubuntu", "Production", "Gate One"],
  updateSequenceNum: 11,
  updated: 1356974567922,
  url: "ssh://localhost",
  visits: 0
}
```

Most of that should be self-explanatory except the `updateSequenceNum` (aka USN). The USN value is used to determine when this bookmark was last changed in comparison to the highest USN stored on the server. By comparing the highest client-side USN against the highest server-side USN we can determine what (if anything) has changed since the last synchronization. It is much more efficient than enumerating all bookmarks on both the client and server in order to figure out what's different.

`GateOne.Bookmarks.createBookmark(bmContainer, bookmark, delay)`

Creates a new bookmark element and places it in *bmContainer*. Also returns the bookmark element.

Arguments

- **bmContainer** (*DOM_node*) – The DOM node we're going to be placing the bookmark.
- **bookmark** (*object*) – A bookmark object (presumably taken from `GateOne.Bookmarks.bookmarks`)
- **delay** (*number*) – The amount of milliseconds to wait before translating (sliding) the bookmark into view.

- **ad** (*boolean*) – If true, will not bother adding tags or edit/delete/share links.

GateOne.Bookmarks.**createOrUpdateBookmark**(*obj*)

Creates or updates a bookmark (in GateOne.Bookmarks.bookmarks and localStorage) using the given bookmark *obj*.

GateOne.Bookmarks.**createPanel**(*[embedded]*)

Creates the bookmarks panel. If the bookmarks panel already exists it will be destroyed and re-created, resetting the pagination.

If *embedded* is true then we'll just load the header (without search).

GateOne.Bookmarks.**createSortOpts**()

Returns a span representing the current sort direction and "sort by" type.

GateOne.Bookmarks.**deleteBookmark**(*obj*)

Asks the user for confirmation then deletes the chosen bookmark...

obj can either be a URL (string) or the "go_bm_delete" anchor tag.

Note: Not the same thing as GateOne.Bookmarks.removeBookmark().

GateOne.Bookmarks.**deletedBookmarksSyncComplete**(*message*)

Handles the response from the server after we've sent the 'bookmarks_deleted' WebSocket action. Resets localStorage[GateOne.prefs.prefix+'deletedBookmarks'] and displays a message to the user indicating how many bookmarks were just deleted.

GateOne.Bookmarks.**editBookmark**(*obj*)

Opens the bookmark editor for the given *obj* (the bookmark element on the page).

Note: Only meant to be called from a 'bm_edit' anchor tag (as the *obj*).

GateOne.Bookmarks.**exportBookmarks**(*[bookmarks]*)

Allows the user to save their bookmarks as a Netscape-style HTML file. Immediately starts the download.

If *bookmarks* is given, that array will be what is exported. Otherwise the complete GateOne.Bookmarks.bookmarks array will be exported.

GateOne.Bookmarks.**filterBookmarksBySearchString**(*str*)

Filters bookmarks to those matching *str* (used by the search function).

GateOne.Bookmarks.**flushIconQueue**()

Loops over localStorage[GateOne.prefs.prefix+'iconQueue'] fetching icons until it is empty.

If the queue is currently being processed this function won't do anything when called.

GateOne.Bookmarks.**generateTip**()

Returns a random, helpful tip for using bookmarks (as a string).

GateOne.Bookmarks.**getAutotags**(*[bookmarks]*)

Returns an array of all the autotags in GateOne.Bookmarks.bookmarks or *bookmarks* if given.

Note: Ordered alphabetically with the URL types coming before date tags.

GateOne.Bookmarks.**getBookmarkObj**(*URL*)

Returns the bookmark object associated with the given *URL*.

GateOne.Bookmarks.**getDateTag**(*dateObj*)

Given a Date object, returns a string such as "<7 days". Suitable for use as an autotag.

GateOne.Bookmarks.**getMaxBookmarks**(*elem*)

Calculates and returns the number of bookmarks that will fit in the given element (*elem*). *elem* may be an element ID or a DOM node object.

GateOne.Bookmarks.**getTags**(*[bookmarks]*)

Returns an array of all the tags in GateOne.Bookmarks.bookmarks or *bookmarks* if given.

Note: Ordered alphabetically

GateOne.Bookmarks.**highestUSN**()

Returns the highest updateSequenceNum that exists in all bookmarks.

GateOne.Bookmarks.**httpIconHandler**(*bookmark*)

Retrieves the icon for the given HTTP or HTTPS *bookmark* and saves it in the bookmarks DB.

GateOne.Bookmarks.**incrementVisits**(*URL*)

Increments by 1 the 'visits' value of the bookmark object associated with the given *URL*.

GateOne.Bookmarks.**init**()

Creates the bookmarks panel, initializes some important variables, registers the Control-Alt-b keyboard shortcut, and registers the following WebSocket actions:

```
GateOne.Net.addAction('terminal:bookmarks_updated', GateOne.Bookmarks.syncBookmarks);
GateOne.Net.addAction('terminal:bookmarks_save_result', GateOne.Bookmarks.syncComplete);
GateOne.Net.addAction('terminal:bookmarks_delete_result', GateOne.Bookmarks.deletedBookmarksSyncComplete);
GateOne.Net.addAction('terminal:bookmarks_renamed_tags', GateOne.Bookmarks.tagRenameComplete);
```

GateOne.Bookmarks.**loadBookmarks**(*[delay]*)

Filters/sorts/displays bookmarks and updates the bookmarks panel to reflect the current state of things (draws the tag cloud and ensures the pagination is correct).

If *delay* (milliseconds) is given, loading of bookmarks will be delayed by that amount before they're drawn (for animation purposes).

GateOne.Bookmarks.**loadPagination**(*bookmarks*, *page*)

Sets up the pagination for the given array of bookmarks and returns the pagination node.

If *page* is given the pagination will highlight the given page number and adjust the prev/next links accordingly.

GateOne.Bookmarks.**loadTagCloud**(*[active]*)

Loads the tag cloud. If *active* is given it must be one of 'tags' or 'autotags'. It will mark the appropriate header as inactive and load the respective tags.

GateOne.Bookmarks.**openBookmark**(*URL*)

Calls the function in GateOne.Bookmarks.URLHandlers associated with the protocol of the given *URL*.

If no function is registered for the given *URL* protocol a new browser window will be opened using the given *URL*.

GateOne.Bookmarks.**openExportDialog**()

Creates a dialog where the user can select some options and export their bookmarks.

GateOne.Bookmarks.**openImportDialog**()

Displays the form where a user can create or edit a bookmark.

If *URL* is given, pre-fill the form with the associated bookmark for editing.

GateOne.Bookmarks.**openNewBookmarkForm**(*[URL]*)

Displays the form where a user can create or edit a bookmark.

If *URL* is given, pre-fill the form with the associated bookmark for editing.

`GateOne.Bookmarks.openRenameDialog(tagName)`

Creates a dialog where the user can rename the given *tagName*.

`GateOne.Bookmarks.openSearchDialog(URL, title)`

Creates a dialog where the user can utilize a keyword search *URL*. *title* will be used to create the dialog title like this: "Keyword Search: *title*".

`GateOne.Bookmarks.panelToggleIn(panel)`

Called when `panel_toggle` event is triggered, calls `GateOne.Bookmarks.createPanel()` if *panel* is the Bookmarks panel.

`GateOne.Bookmarks.registerIconHandler(protocol, handler)`

Registers the given *handler* as the function to use whenever a bookmark icon needs to be retrieved for the given *protocol*.

When the given *handler* is called it will be passed the bookmark object as the only argument. It is up to the handler to call (or not) `GateOne.Bookmarks.storeFavicon(bookmark, <icon data URI>)`; to store the icon.

`GateOne.Bookmarks.registerURLHandler(protocol, handler)`

Registers the given *handler* as the function to use whenever a bookmark is opened with a matching *protocol*.

When the given *handler* is called it will be passed the URL as the only argument.

`GateOne.Bookmarks.removeBookmark(url[, callback])`

Removes the bookmark matching *url* from `GateOne.Bookmarks.bookmarks` and saves the change to `localStorage`.

If *callback* is given it will be called after the bookmark has been deleted.

Note: Not the same thing as `GateOne.Bookmarks.deleteBookmark()`.

`GateOne.Bookmarks.removeFilterDateTag(bookmarks, dateTag)`

Removes the given *dateTag* from the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.removeFilterTag(bookmarks, tag)`

Removes the given *tag* from the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.removeFilterURLTypeTag(bookmarks, typeTag)`

Removes the given *typeTag* from the filter list. *bookmarks* is unused.

`GateOne.Bookmarks.renameTag(oldName, newName)`

Renames the tag matching *oldName* to be *newName* for all bookmarks that have it.

`GateOne.Bookmarks.sortFunctions`

An associative array of functions that are used to sort bookmarks. When the user clicks on one of the sorting options it assigns one of these functions to `GateOne.Bookmarks.sortfunc()` which is then applied like so:

```
bookmarks.sort(GateOne.Bookmarks.sortfunc);
```

`GateOne.Bookmarks.sortFunctions.alphabetical(a, b)`

Sorts bookmarks alphabetically.

`GateOne.Bookmarks.sortFunctions.created(a, b)`

Sorts bookmarks by date modified followed by alphabetical.

`GateOne.Bookmarks.sortFunctions.visits(a, b)`

Sorts bookmarks according to the number of visits followed by alphabetical.

`GateOne.Bookmarks.storeBookmark(bookmarkObj[, callback])`

Stores the given *bookmarkObj* in `localStorage`.

If *callback* is given it will be executed after the bookmark is stored with the *bookmarkObj* as the only argument.

`GateOne.Bookmarks.storeBookmarks(bookmarks[, recreatePanel[, skipTags]])`

Takes an array of *bookmarks* and stores them in `GateOne.Bookmarks.bookmarks`.

If *recreatePanel* is true, the panel will be re-drawn after bookmarks are stored.

If *skipTags* is true, bookmark tags will be ignored when saving *bookmarks*.

`GateOne.Bookmarks.storeFavicon(bookmark, dataURI)`

Stores the given *dataURI* as the 'favicon' image for the given *bookmark*.

Note: *dataURI* must be pre-encoded data:URI

`GateOne.Bookmarks.syncBookmarks(response)`

Called when the `terminal:bookmarks_updated` WebSocket action is received from the server. Removes bookmarks marked as deleted on the server, uploads new bookmarks that are not on the server (yet), and processes any tags that have been renamed.

`GateOne.Bookmarks.syncComplete(response)`

Called when the sync (download) is completed. Stores the current highest `updateSequenceNum` in `localStorage`, and notifies the user of any errors that occurred during synchronization.

`GateOne.Bookmarks.tagContextMenu(elem)`

Called when we right-click on a tag *elem*. Gives the user the option to rename the tag or cancel the context menu.

`GateOne.Bookmarks.tagRenameComplete(result)`

Called when the 'bookmarks_renamed_tags' WebSocket action is received from the server. Deletes `localStorage[GateOne.prefs.prefix+'renamedTags']` (which stores tags that have been renamed and are awaiting sync) and displays a message to the user indicating that tags were renamed successfully.

`GateOne.Bookmarks.toggleSortOrder()`

Reverses the order of the bookmarks list.

`GateOne.Bookmarks.updateIcon(bookmark)`

Calls the handler in `GateOne.Bookmarks.iconHandlers` associated with the given *bookmark.url* protocol.

If no handler can be found no operation will be performed.

`GateOne.Bookmarks.updateIcons(urls)`

Loops over the given *urls* attempting to fetch and store their respective favicons.

Note: This function is only used when debugging. It is called by no other functions.

`GateOne.Bookmarks.updateProgress(name, total, num[, desc])`

Creates/updates a progress bar given a *name*, a *total*, and *num* representing the current state of an action.

Optionally, a description (*desc*) may be provided that will be placed above the progress bar.

`GateOne.Bookmarks.updateUSN(obj)`

Updates the `updateSequenceNum` of the bookmark matching *obj* in `GateOne.Bookmarks.bookmarks` (and in `localStorage` via `storeBookmark()`).

GateOne.Bookmarks.userLoginSync(*username*)

This gets attached to the go:js_loaded event. Calls the server-side terminal:bookmarks_get WebSocket action with the current USN (Update Sequence Number) to ensure the user's bookmarks are in sync with what's on the server.

Python bookmarks.py - A plugin for Gate One that adds fancy bookmarking capabilities.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
    'Web': [
        (r"/bookmarks/fetchicon", FaviconHandler),
        (r"/bookmarks/export", ExportHandler),
        (r"/bookmarks/import", ImportHandler),
    ],
    'WebSocket': {
        'terminal:bookmarks_sync': save_bookmarks,
        'terminal:bookmarks_get': get_bookmarks,
        'terminal:bookmarks_deleted': delete_bookmarks,
        'terminal:bookmarks_rename_tags': rename_tags,
    },
    'Events': {
        'terminal:authenticate': send_bookmarks_css_template
    }
}
```

Docstrings

bookmarks.unescape(*s*)

Unescape HTML code refs; c.f. <http://wiki.python.org/moin/EscapingHtml>

bookmarks.parse_bookmarks_html(*html*)

Reads the Netscape-style bookmarks.html in string, *html* and returns a list of Bookmark objects.

bookmarks.get_json_tags(*bookmarks*, *url*)

Iterates over *bookmarks* (dict) trying to find tags associated with the given *url*. Returns the tags found as a list.

bookmarks.get_ns_json_bookmarks(*json_dict*, *bookmarks*)

Given a *json_dict*, updates *bookmarks* with each URL as it is found within.

Note: Only works with Netscape-style bookmarks.json files.

bookmarks.parse_bookmarks_json(*json_str*)

Given *json_str*, returns a list of bookmark objects representing the data contained therein.

class bookmarks.BookmarksDB(*user_dir*, *user*)

Used to read and write bookmarks to a file on disk. Can also synchronize a given list of bookmarks with what's on disk. Uses a given bookmark's updateSequenceNum to track what wins the "who is newer?" comparison.

Sets up our bookmarks database object and reads everything in.

open_bookmarks()

Opens the bookmarks stored in self.user_dir. If not present, an empty file will be created.

save_bookmarks()

Saves self.bookmarks to self.bookmarks_path as a JSON-encoded list.

sync_bookmarks(*bookmarks*)

Given *bookmarks*, synchronize with self.bookmarks doing conflict resolution and whatnot.

delete_bookmark(*bookmark*)

Deletes the given *bookmark*.

get_bookmarks(*updateSequenceNum=0*)

Returns a list of bookmarks newer than *updateSequenceNum*. If *updateSequenceNum* is 0 or undefined, all bookmarks will be returned.

get_highest_USN()

Returns the highest updateSequenceNum in self.bookmarks

rename_tag(*old_tag*, *new_tag*)

Goes through all bookmarks and renames all tags named *old_tag* to be *new_tag*.

class bookmarks.**FaviconHandler**(*application*, *request*, ***kwargs*)

Retrieves the biggest favicon-like icon at the given URL. It will try to fetch apple-touch-icons (which can be nice and big) before it falls back to grabbing the favicon.

Note: Works with GET and POST requests but POST is preferred since it keeps the URL from winding up in the server logs.

get_favicon_url(*html*)

Parses *html* looking for a favicon URL. Returns a tuple of:

(*<url>*, *<mimetype>*)

If no favicon can be found, returns:

(*None*, *None*)

icon_multifetch(*urls*, *response*)

Fetches the icon at the given URLs, stopping when it finds the biggest. If an icon is not found, calls itself again with the next icon URL. If the icon is found, writes it to the client and finishes the request.

icon_fetch(*url*, *mimetype*, *response*)

Returns the fetched icon to the client.

class bookmarks.**ImportHandler**(*application*, *request*, ***kwargs*)

Takes a bookmarks.html in a POST and returns a list of bookmarks in JSON format

class bookmarks.**ExportHandler**(*application*, *request*, ***kwargs*)

Takes a JSON-encoded list of bookmarks and returns a Netscape-style HTML file.

bookmarks.**send_bookmarks_css_template**(*self*)

Sends our bookmarks.css template to the client using the 'load_style' WebSocket action. The rendered template will be saved in Gate One's 'cache_dir'.

The Convenience Plugin

JavaScript

GateOne.Convenience

Provides numerous syntax highlighting functions and conveniences to provide quick and useful information to the user.

GateOne.Convenience.**IPInfo**(*elem*)

Calls GateOne.SSH.execRemoteCmd(*term*, 'host ' + elem.innerHTML).

GateOne.Convenience.**addPrefs()**

Adds a number of configurable elements to Gate One's preferences panel.

GateOne.Convenience.**displayGroupInfo(output)**

Displays a message containing all of the group's information using GateOne.Convenience.groupTemp and by parsing the output of the `getent passwd | grep ...` command.

GateOne.Convenience.**displayIPInfo(output)**

Displays the result of host.

GateOne.Convenience.**displayUserInfo(output)**

Parses the output of the `'getent passwd <user>'` command and displays it in an easy-to-read format.

GateOne.Convenience.**groupInfo(elem)**

Calls `'getent group' + elem.innerHTML` on the server using GateOne.SSH.execRemoteCmd(). The result will be handled by GateOne.Convenience.displayGroupInfo()

GateOne.Convenience.**groupInfo2(output)**

Parses the output of the `getent group` command, saves the details as HTML in GateOne.Convenience.groupTemp, then calls `getent passwd` looking for all the users that have the given group as their primary GID. The final output will be displayed via GateOne.Convenience.displayGroupInfo().

GateOne.Convenience.**groupInfoError(result)**

Displays a message indicating there was an error getting info on the group.

GateOne.Convenience.**init()**

Sets up all our conveniences.

GateOne.Convenience.**permissionsBitmask(elemOrString)**

Returns the bitmask (i.e. `chmod <bitmask>`) to a file/directory's permissions. For example:

```
>>> someElement.innerHTML = 'drwxrwxr-x';
>>> GateOne.Convenience.permissionsBitmask(someElement);
'0775'
```

GateOne.Convenience.**permissionsInfo(elem)**

Displays information about the `'ls -l'` permissions contained within *elem*. *elem.innerHTML* **must** be something like `'drwxrwxr-x'`.

GateOne.Convenience.**registerIPConvenience()**

Registers a text transform that makes IPv4 addresses into spans that will execute host when clicked.

Note: This feature will disable itself if the SSH plugin is disabled/missing.

GateOne.Convenience.**registerIPConvenience()**

Removes all the text transforms that apply to IP addresses.

GateOne.Convenience.**registerLSConvenience()**

Registers a number of text transforms to add conveniences to the output of `'ls -l'`.

GateOne.Convenience.**registerPSConvenience()**

Registers a text transform that adds syntax highlighting to the output of `'ps'`.

GateOne.Convenience.**registerSyslogConvenience()**

Registers a text transform that makes standard syslog output easier on the eyes.

GateOne.Convenience.**savePrefsCallback**()

Called when the user clicks the “Save” button in the prefs panel.

GateOne.Convenience.**toggleBackground**(*result*)

Toggles a background color on and off for the given *elem* by adding or removing the ‘select-edrow’ class.

GateOne.Convenience.**unregisterLSConvenience**()

Removes all of the text transforms that apply to the output of ‘ls -l’

GateOne.Convenience.**unregisterPSConvenience**()

Removes all the text transforms associated with ps output.

GateOne.Convenience.**unregisterSyslogConvenience**()

Removes all the text transforms associated with syslog output.

GateOne.Convenience.**userInfo**(*elem*)

Calls ‘getent passwd’ + *elem*.innerHTML on the server using GateOne.SSH.execRemoteCmd(). The result will be handled by GateOne.Convenience.displayUserInfo()

GateOne.Convenience.**userInfoError**(*result*)

Displays a message indicating there was an error getting info on the user.

The Example Plugin

JavaScript example.js - The client-side portion of Gate One’s Example plugin.

GateOne.Terminal.Example.**generateAuthObject**(*api_key*, *secret*, *upn*)

Returns a properly-constructed authentication object that can be used with Gate One’s API authentication mode. The timestamp, signature, signature_method, and api_version values will be created automatically.

Arguments

- **api_key** (*string*) – The API key to use when generating the authentication object. Must match Gate One’s api_keys setting (e.g. in server.conf).
- **secret** (*string*) – The secret attached to the given *api_key*.
- **upn** (*string*) – The userPrincipalName (aka UPN or username) you’ll be authenticating.

Note: This will also attach an ‘example_attribute’ that will be automatically assigned to the ‘user’ dict on the server so it can be used for other purposes (e.g. authorization checks and inside of plugins).

GateOne.Terminal.Example.**init**()

The init() function of every JavaScript plugin gets called automatically after the WebSocket is connected is authenticated.

The Example plugin’s init() function sets up some internal variables, keyboard shortcuts (Control-Alt-L to open the load graph), and adds some buttons to the Info & Tools menu.

GateOne.Terminal.Example.**stopGraph**(*result*)

Clears the GateOne.Terminal.Example.graphUpdateTimer, removes the canvas element, and stops the smoothie graph streaming. *result* is unused.

GateOne.Terminal.Example.**toggleLoadGraph**(*term*)

Displays a real-time load graph of the given terminal (inside of it as a GateOne.Visual.widget()).

GateOne.Terminal.Example.**topTop**(*term*)

Displays the top three CPU-hogging processes on the server in real-time (updating every three seconds just like top).

GateOne.Terminal.Example.**updateGraph**(*output*)

Updates GateOne.Terminal.Example.line1 through line3 by parsing the *output* of the 'uptime' command.

GateOne.Terminal.Example.**updateTop**(*output*)

Updates the GateOne.Terminal.Example.topTop() output on the screen when we receive *output* from the Gate One server. Here's what the output should look like:

```
PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
  1 root        20   0 24052 2132 1316  S   0.0   0.4   0:00.35 /sbin/init
  2 root        20   0     0     0     0  S   0.0   0.0   0:00.00 [kthreadd]
  3 root        20   0     0     0     0  S   0.0   0.0   0:00.08 [ksoftirqd/0]
```

GateOne.Terminal.Terminal.Example.**stopTop**(*result*)

Clears the GateOne.Terminal.Example.topUpdateTimer and removes the 'toptop' element. *result* is unused.

Python example.py - A plugin to demonstrate how to write a Python plugin for Gate One. Specifically, how to write your own web handlers, WebSocket actions, and take advantage of all the available hooks and built-in functions.

Tip: This plugin is heavily commented with useful information. Click on the [source] links to the right of any given class or function to see the actual code.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
    'Web': [(r"/example", ExampleHandler)],
    'WebSocket': {
        'example_action': example_websocket_action
    },
    'Escape': example_opt_esc_handler,
}
```

Docstrings

class example.ExampleHandler(*application, request, **kwargs*)

This is how you add a URL handler to Gate One... This example attaches itself to <https://<your Gate One server>/example> in the 'Web' hook at the bottom of this file. It works just like any Tornado RequestHandler. See:

<http://www.tornadoweb.org/documentation/web.html>

...for documentation on how to write a tornado.web.RequestHandler for the Tornado framework. Fairly boilerplate stuff.

Note: The only reason we use gateone.BaseHandler instead of a vanilla tornado.web.RequestHandler is so we have access to Gate One's gateone.BaseHandler.get_current_user() function.

get(**args, **kwargs*)

Handle an HTTP GET request to this RequestHandler. Connect to:

<https://<your Gate One server>/example>

...to try it out.

post()

Example Handler for an **HTTP POST** request. Doesn't actually do anything.

example.**example_websocket_action**(*self, message*)

This **WebSocket** action gets exposed to the client automatically by way of the 'WebSocket' hook at the bottom of this file. The way it works is like this:

How The WebSocket Hook Works

Whenever a message is received via the **WebSocket** Gate One will automatically decode it into a Python dict (only JSON-encoded messages are accepted). Any and all keys in that dict will be assumed to be 'actions' (just like `GateOne.Net.actions` but on the server) such as this one. If the incoming key matches a registered action that action will be called like so:

```
key(value)
# ...or just:
key() # If the value is None ('null' in JavaScript)
```

...where *key* is the action and *value* is what will be passed to said action as an argument. Since Gate One will automatically decode the message as JSON the *value* will typically be passed to actions as a single dict. You can provide different kinds of arguments of course but be aware that their ordering is unpredictable so always be sure to either pass *one* argument to your function (assuming it is a dict) or 100% keyword arguments.

The *self* argument here is automatically assigned by `TerminalApplication` using the `utils.bind` method.

The typical naming convention for **WebSocket** actions is: `<plugin name>_`. Whether or not your action names match your function names is up to you. All that matters is that you line up an *action* (string) with a *function* in `hooks['WebSocket']` (see below).

This **WebSocket** *action* duplicates the functionality of Gate One's built-in `gateone.TerminalWebSocket.pong()` function. You can see how it is called by the client (browser) inside of `example.js` (which is in this plugin's 'static' dir).

example.**example_opt_esc_handler**(*self, message, term=None, multiplex=None*)

Gate One includes a mechanism for plugins to send messages from terminal programs directly to plugins written in Python. It's called the "Special Optional Escape Sequence Handler" or SOESH for short. Here's how it works: Whenever a terminal program emits, `"x1b]_;"` it gets detected by Gate One's `Terminal` class (which lives in `terminal.py`) and it will execute whatever callback is registered for SOESH. Inside of Gate One this callback will always be `gateone.TerminalWebSocket.esc_opt_handler()`.

example.**example_command_hook**(*self, command, term=None*)

This demonstrates how to modify Gate One's configured 'command' before it is executed. It will replace any occurrence of `%EXAMPLE%` with 'foo'. So if `command = "some_script.sh %EXAMPLE%"` in your `server.conf` it would be transformed to `"some_script.sh foo"` before being executed when a user opens a new terminal.

The Logging Plugin

JavaScript `logging.js` - The client-side portion of Gate One's Logging plugin.

GateOne.TermLogging.**createLogItem**(*container*, *logObj*, *delay*)

Creates a logItem element using *logObj* and places it in *container*.

delay controls how long it will wait before using a CSS3 effect to move it into view.

GateOne.TermLogging.**createPanel**()

Creates the logging panel (just the empty shell of it).

GateOne.TermLogging.**displayFlatLogAction**(*message*)

Opens a new window displaying the (flat) log contained within *message* if there are no errors reported.

GateOne.TermLogging.**displayMetadata**(*logFile*)

Displays the information about the log file, *logFile* in the metadata area of the log viewer.

GateOne.TermLogging.**displayPlaybackLogAction**(*message*)

Opens a new window playing back the log contained within *message* if there are no errors reported.

GateOne.TermLogging.**getMaxLogItems**(*elem*)

Calculates and returns the number of log items that will fit in the given element (*elem*). *elem* may be a DOM node or an element ID (string).

GateOne.TermLogging.**incomingLogAction**(*message*)

Adds *message*['log'] to GateOne.TermLogging.serverLogs and places it into the view.

GateOne.TermLogging.**incomingLogsCompleteAction**(*message*)

Sets the header of the log viewer and displays a message to indicate we're done loading.

GateOne.TermLogging.**init**()

Creates the log viewer panel and registers the following WebSocket actions:

```
GateOne.Net.addAction('terminal:logging_log', GateOne.TermLogging.incomingLogAction);
GateOne.Net.addAction('terminal:logging_logs_complete', GateOne.TermLogging.incomingLogsCompleteAction);
GateOne.Net.addAction('terminal:logging_log_flat', GateOne.TermLogging.displayFlatLogAction);
GateOne.Net.addAction('terminal:logging_log_playback', GateOne.TermLogging.displayPlaybackLogAction);
```

GateOne.TermLogging.**loadLogs**(*forceUpdate*)

After GateOne.TermLogging.serverLogs has been populated this function will redraw the view depending on sort and pagination values.

If *forceUpdate* empty out GateOne.TermLogging.serverLogs and tell the server to send us a new list.

GateOne.TermLogging.**loadPagination**(*logItems*[, *page*])

Sets up the pagination for the given array of *logItems* and returns the pagination node.

If *page* is given, the pagination will highlight the given page number and adjust prev/next accordingly.

GateOne.TermLogging.**openLogFlat**(*logFile*)

Tells the server to open *logFile* for playback via the 'terminal:logging_get_log_flat' server-side WebSocket action (will end up calling displayFlatLogAction()).

GateOne.TermLogging.**openLogPlayback**(*logFile*[, *where*])

Tells the server to open *logFile* for playback via the 'terminal:logging_get_log_playback' server-side WebSocket action (will end up calling displayPlaybackLogAction()).

If *where* is given and it is set to 'preview' the playback will happen in the log_preview iframe.

GateOne.TermLogging.**saveRenderedLog**(*logFile*)

Tells the server to open *logFile* rendered as a self-contained recording (via the 'logging_get_log_file' WebSocket action) and send it back to the browser for saving (using the 'save_file' WebSocket action).

GateOne.TermLogging.**sessionLoggingDisabled()**

Just removes the “Log Viewer” button from the Terminal info panel. It gets sent if the server has "session_logging": false.

GateOne.TermLogging.**sortFunctions**

An associative array of functions that are used to sort logs. When the user clicks on one of the sorting options it assigns one of these functions to GateOne.TermLogging.sortfunc() which is then applied like so:

```
logs.sort(GateOne.TermLogging.sortfunc);
```

GateOne.TermLogging.sortFunctions.**alphabetical(a, b)**

Sorts logs alphabetically using the title (connect_string).

GateOne.TermLogging.sortFunctions.**alphabetical(a, b)**

Sorts logs according to their size.

GateOne.TermLogging.sortFunctions.**date(a, b)**

Sorts logs by date (start_date) followed by alphabetical order of the title (connect_string).

GateOne.TermLogging.**toggleSortOrder()**

Reverses the order of the logs array.

Python logging_plugin.py - A plugin for Gate One that provides logging-related functionality.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
    'WebSocket': {
        'logging_get_logs': enumerate_logs,
        'logging_get_log_flat': retrieve_log_flat,
        'logging_get_log_playback': retrieve_log_playback,
        'logging_get_log_file': save_log_playback,
    },
    'Events': {
        'terminal:authenticate': send_logging_css_template
    }
}
```

Docstrings

logging_plugin.**get_256_colors(self)**

Returns the rendered 256-color CSS.

logging_plugin.**enumerate_logs(self, limit=None)**

Calls _enumerate_logs() via a multiprocessing.Process so it doesn't cause the IOLoop to block.

Log objects will be returned to the client one at a time by sending 'logging_log' actions to the client over the WebSocket (self).

logging_plugin.**_enumerate_logs(queue, user, users_dir, limit=None)**

Enumerates all of the user's logs and sends the client a "logging_logs" message with the result.

If limit is given, only return the specified logs. Works just like MySQL: limit="5,10" will retrieve logs 5-10.

logging_plugin.**retrieve_log_flat(self, settings)**

Calls _retrieve_log_flat() via a multiprocessing.Process so it doesn't cause the IOLoop to block.

Parameters **settings** (*dict*) – A dict containing the *log_filename*, *colors*, and *theme* to use when generating the HTML output.

Here's the details on *settings*:

Parameters

- **settings['log_filename']** – The name of the log to display.
- **settings['colors']** – The CSS color scheme to use when generating output.
- **settings['theme']** – The CSS theme to use when generating output.
- **settings['where']** – Whether or not the result should go into a new window or an iframe.

`logging_plugin._retrieve_log_flat(queue, settings)`

Writes the given *log_filename* to *queue* in a flat format equivalent to:

```
./logviewer.py --flat log_filename
```

settings - A dict containing the *log_filename*, *colors_css*, and *theme_css* to use when generating the HTML output.

`logging_plugin.retrieve_log_playback(self, settings)`

Calls `_retrieve_log_playback()` via a `multiprocessing.Process` so it doesn't cause the `IOLoop` to block.

`logging_plugin._retrieve_log_playback(queue, settings)`

Writes a JSON-encoded message to the client containing the log in a self-contained HTML format similar to:

```
./logviewer.py log_filename
```

settings - A dict containing the *log_filename*, *colors*, and *theme* to use when generating the HTML output.

Parameters

- **settings['log_filename']** – The name of the log to display.
- **settings['colors_css']** – The CSS color scheme to use when generating output.
- **settings['theme_css']** – The entire CSS theme `<style>` to use when generating output.
- **settings['where']** – Whether or not the result should go into a new window or an iframe.

The output will look like this:

```
{
  'result': "Success",
  'log': <HTML rendered output>,
  'metadata': {<metadata of the log>}
}
```

It is expected that the client will create a new window with the result of this method.

`logging_plugin.save_log_playback(self, settings)`

Calls `_save_log_playback()` via a `multiprocessing.Process` so it doesn't cause the `IOLoop` to block.

`logging_plugin._save_log_playback(queue, settings)`

Writes a JSON-encoded message to the client containing the log in a self-contained HTML format similar to:

```
./logviewer.py log_filename
```

The difference between this function and `_retrieve_log_playback()` is that this one instructs the client to save the file to disk instead of opening it in a new window.

Parameters

- **settings['log_filename']** – The name of the log to display.
- **settings['colors']** – The CSS color scheme to use when generating output.
- **settings['theme']** – The CSS theme to use when generating output.
- **settings['where']** – Whether or not the result should go into a new window or an iframe.

The output will look like this:

```
{
  'result': "Success",
  'data': <HTML rendered output>,
  'mimetype': 'text/html'
  'filename': <filename of the log recording>
}
```

It is expected that the client will create a new window with the result of this method.

`logging_plugin.session_logging_check(self)`

Attached to the `terminal:session_logging_check` WebSocket action; replies with `terminal:logging_sessions_disabled` if terminal session logging is disabled.

Note: The `terminal:logging_sessions_disabled` message just has the client remove the “Log Viewer” button so they don’t get confused with an always-empty log viewer.

`logging_plugin.send_logging_css_template(self)`

Sends our `logging.css` template to the client using the ‘load_style’ WebSocket action. The rendered template will be saved in Gate One’s ‘cache_dir’.

The Notice Plugin

Python `notice.py` - A plugin for Gate One that adds an escape sequence handler that will tell the client browser to display a message whenever said escape sequence is encountered. Any terminal program can emit the following escape sequence to display a message in the browser:

```
\x1b[;notice|<the message>\x07
```

Very straightforward and also very powerful.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
  'Escape': notice_esc_seq_handler,
}
```

Docstrings

`notice.notice_esc_seq_handler(self, message, term=None, multiplex=None)`

Handles text passed from the special optional escape sequence handler to display a *message* to the connected client (browser). It can be invoked like so:

```
$ echo -e "\033]_;notice|Text passed to some_function()\\007"
```

See also:

`app_terminal.TerminalApplication.opt_esc_handler` and `terminal.Terminal._opt_handler()`

The Playback Plugin

JavaScript `playback.js` - The client-side portion of Gate One's Playback plugin.

`GateOne.Playback.addControls()`

Adds the session playback controls to Gate One's element (`GateOne.prefs.goDiv`).

Note: Will not add playback controls if they're already present.

`GateOne.Playback.hideControls()`

Hides the playback controls.

`GateOne.Playback.init()`

Adds the playback controls to Gate One and adds some GUI elements to the Tools & Info panel. Also attaches the following events/functions:

```
// Add our callback that adds an extra newline to all terminals
GateOne.Events.on("terminal:new_terminal", GateOne.Playback.newTerminalCallback);
// This makes sure our playback frames get added to the terminal object whenever the screen is updated
GateOne.Events.on("terminal:term_updated", GateOne.Playback.pushPlaybackFrame);
// This makes sure our prefs get saved along with everything else
GateOne.Events.on("go:save_prefs", GateOne.Playback.savePrefsCallback)
// Hide the playback controls when in grid view
GateOne.Events.on("go:grid_view:open", GateOne.Playback.hideControls);
// Show the playback controls when no longer in grid view
GateOne.Events.on("go:grid_view:close", GateOne.Playback.showControls);
```

`GateOne.Playback.newTerminalCallback(term, calledTwice)`

This gets added to the 'terminal:new_terminal' event to ensure that there's some extra space at the bottom of each terminal to make room for the playback controls.

It also calls `GateOne.Playback.addControls()` to make sure they're present only after a new terminal is open.

`GateOne.Playback.playPauseControl(e)`

Toggles play/pause inside the current terminal. Meant to be attached to the Play/Pause icon's onclick event.

`GateOne.Playback.playbackRealtime(term)`

Plays back the given terminal's session one frame at a time. Meant to be used inside of an interval timer.

`GateOne.Playback.pushPlaybackFrame(term)`

Adds the current screen of *term* to `GateOne.Terminal.terminals[term].playbackFrames`.

`GateOne.Playback.savePrefsCallback()`

Called when the user clicks the "Save" button in the prefs panel. Makes sure the 'playbackFrames' setting gets updated according to what the user entered into the form.

`GateOne.Playback.saveRecording(term)`

Saves the session playback recording by sending the given *term*'s 'playbackFrames' to the server to have them rendered.

When the server is done rendering the recording it will be sent back to the client via the 'save_file' WebSocket action.

`GateOne.Playback.selectFrame(term, ms)`

For the given *term*, returns the last frame # with a 'time' less than the first frame's time + *ms*.

`GateOne.Playback.showsControls()`

Shows the playback controls again after they've been hidden via `GateOne.Playback.hideControls()`.

`GateOne.Playback.startPlayback(term)`

Plays back the given terminal's session in real-time.

`GateOne.Playback.switchWorkspaceEvent(workspace)`

Called whenever Gate One switches to a new workspace; checks whether or not this workspace is home to a terminal and if so, shows the playback controls.

If no terminal is present in the current workspace the playback controls will be removed.

`GateOne.Playback.updateClock([dateObj])`

Updates the clock with the time in the given *dateObj*.

If no *dateObj* is given, the clock will be updated with the current local time.

Python `playback.py` - A plugin for Gate One that adds support for saving and playing back session recordings.

Note: Yes this only contains one function and it is exposed to clients through a WebSocket hook.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
    'WebSocket': {
        'playback_save_recording': save_recording,
    }
}
```

Docstrings

`playback.get_256_colors(self)`

Returns the rendered 256-color CSS.

`playback.save_recording(self, settings)`

Handles uploads of session recordings and returns them to the client in a self-contained HTML file that will auto-start playback.

..note:: The real crux of the code that handles this is in the template.

The SSH Plugin

JavaScript `ssh.js` - The client-side portion of Gate One's SSH plugin.

`GateOne.SSH.autoConnect()`

Automatically connects to `GateOne.prefs.autoConnectURL` if it set.

GateOne.SSH.**bookmarkIconHandler**(*bookmark*)

Saves the GateOne.Icons.SSH icon in the given bookmark using GateOne.Bookmarks.storeFavicon().

Note: This gets registered for the 'ssh' and 'telnet' inside of GateOne.SSH.postInit().

GateOne.SSH.**commandCompleted**(*message*)

Uses the contents of *message* to report the results of the command executed via execRemoteCmd().

The *message* should be something like:

```
{
  'term': 1,
  'cmd': 'uptime',
  'output': ' 20:45:27 up 13 days,  3:44,  9 users,  load average: 1.21, 0.79, 0.57',
  'result': 'Success'
}
```

If 'result' is anything other than 'Success' the error will be displayed to the user.

If a callback was registered in GateOne.SSH.remoteCmdCallbacks[term] it will be called like so:

```
callback(message['output'])
```

Otherwise the output will just be displayed to the user. After the callback has executed it will be removed from GateOne.SSH.remoteCmdCallbacks.

GateOne.SSH.**connect**(*URL*)

Connects to the given SSH *URL*.

If the current terminal is sitting at the SSH Connect prompt it will be used to make the connection. Otherwise a new terminal will be opened.

GateOne.SSH.**createKHPanel**()

Creates a panel where the user can edit their known_hosts file and appends it to '#gateone'.

If the panel already exists its contents will be destroyed and re-created.

GateOne.SSH.**createPanel**()

Creates the SSH identity management panel (the shell of it anyway).

GateOne.SSH.**deleteCompleteAction**(*message*)

Called when an identity is deleted, calls GateOne.SSH.loadIDs()

GateOne.SSH.**displayHostFingerprint**(*message*)

Displays the host's key as sent by the server via the 'sshjs_display_fingerprint' WebSocket action.

The fingerprint will be colorized using the hex values of the fingerprint as the color code with the last value highlighted in bold.

GateOne.SSH.**displayMetadata**(*identity*)

Displays the information about the given *identity* (its name) in the SSH identities metadata area (on the right). Also displays the buttons that allow the user to delete the identity or upload a certificate.

GateOne.SSH.**displayMetadata**(*container*, *IDObj*, *delay*)

Creates an SSH identity element using *IDObj* and places it into *container*.

delay controls how long it will wait before using a CSS3 effect to move it into view.

GateOne.SSH.**duplicateSession**(*term*)

Duplicates the SSH session at *term* in a new terminal.

GateOne.SSH.**enterPassphraseAction**(*settings*)

Displays the dialog/form where a user can enter a passphrase for a given identity (called by the server if something requires it).

GateOne.SSH.**execRemoteCmd**(*term, command, callback, errorback*)

Executes *command* by creating a secondary shell in the background using the multiplexed tunnel of *term* (works just like `duplicateSession()`).

Calls *callback* when the result of *command* comes back.

Calls *errorback* if there's an error executing the command.

GateOne.SSH.**getConnectString**(*term*)

Asks the SSH plugin on the Gate One server what the SSH connection string is for the given *term*.

GateOne.SSH.**getMaxIDs**(*elem*)

Calculates and returns the number of SSH identities that will fit in the given element ID (*elem*).

GateOne.SSH.**handleConnect**(*connectString*)

Handles the `terminal:sshjs_connect` WebSocket action which should provide an SSH *connectString* in the form of `'user@host:port'`.

The *connectString* will be stored in `GateOne.Terminal.terminals[term]['sshConnectString']` which is meant to be used in duplicating terminals (because you can't rely on the title).

Also requests the host's public SSH key so it can be displayed to the user.

GateOne.SSH.**handleReconnect**(*message*)

Handles the `terminal:sshjs_reconnect` WebSocket action which should provide an object containing each terminal's SSH connection string. Example *message*:

```
{ "term": 1, "connect_string": "user@host1:22" }
```

GateOne.SSH.**incomingIDsAction**(*message*)

This gets attached to the `'sshjs_identities_list'` WebSocket action. Adds *message['identities']* to `GateOne.SSH.identities` and places them into the Identity Manager.

GateOne.SSH.**init**()

Creates the SSH Identity Manager panel, adds some buttons to the Info & Tools panel, and registers the following WebSocket actions & events:

```
GateOne.Net.addAction('terminal:sshjs_connect', GateOne.SSH.handleConnect);
GateOne.Net.addAction('terminal:sshjs_reconnect', GateOne.SSH.handleReconnect);
GateOne.Net.addAction('terminal:sshjs_keygen_complete', GateOne.SSH.keygenComplete);
GateOne.Net.addAction('terminal:sshjs_save_id_complete', GateOne.SSH.saveComplete);
GateOne.Net.addAction('terminal:sshjs_display_fingerprint', GateOne.SSH.displayHostFingerprint);
GateOne.Net.addAction('terminal:sshjs_identities_list', GateOne.SSH.incomingIDsAction);
GateOne.Net.addAction('terminal:sshjs_delete_identity_complete', GateOne.SSH.deleteCompleteAction);
GateOne.Net.addAction('terminal:sshjs_cmd_output', GateOne.SSH.commandCompleted);
GateOne.Net.addAction('terminal:sshjs_ask_passphrase', GateOne.SSH.enterPassphraseAction);
GateOne.Events.on("terminal:new_terminal", GateOne.SSH.getConnectString);
```

GateOne.SSH.**keygenComplete**(*message*)

Called when we receive a message from the server indicating a keypair was generated successfully.

GateOne.SSH.**loadIDs**()

Toggles the SSH Identity Manager into view (if not already visible) and asks the server to send us our list of identities.

GateOne.SSH.**newIDForm**()

Displays the dialog/form where the user can create or edit an SSH identity.

GateOne.SSH.**postInit()**

Registers our 'ssh' and 'telnet' protocol handlers with the Bookmarks plugin.

Note: These things are run inside of the `postInit()` function in order to ensure that `GateOne.Bookmarks` is loaded (and ready-to-go) first.

GateOne.SSH.**saveComplete**(*message*)

Called when we receive a message from the server indicating the uploaded identity was saved.

GateOne.SSH.**updateKH**(*known_hosts*)

Updates the `sshKHTextArea` with the given *known_hosts* file.

Note: Meant to be used as a callback function passed to `GateOne.Utils.xhrGet()`.

GateOne.SSH.**uploadCertificateForm**(*identity*)

Displays the dialog/form where a user can add or replace a certificate associated with their identity.

identity should be the name of the identity associated with this certificate.

GateOne.SSH.**uploadIDForm**()

Displays the dialog/form where a user can upload an SSH identity (that's already been created).

Python `ssh.py` - A plugin for Gate One that adds additional SSH-specific features.

Hooks This Python plugin file implements the following hooks:

```
hooks = {
    'Web': [(r"/ssh", KnownHostsHandler)],
    'WebSocket': {
        'ssh_get_connect_string': get_connect_string,
        'ssh_execute_command': ws_exec_command,
        'ssh_get_identities': get_identities,
        'ssh_get_public_key': get_public_key,
        'ssh_get_private_key': get_private_key,
        'ssh_get_host_fingerprint': get_host_fingerprint,
        'ssh_gen_new_keypair': generate_new_keypair,
        'ssh_store_id_file': store_id_file,
        'ssh_delete_identity': delete_identity,
        'ssh_set_default_identities': set_default_identities
    },
    'Escape': opt_esc_handler,
    'Events': {
        'terminal:authenticate': send_css_template,
        'terminal:authenticate': create_user_ssh_dir
    }
}
```

Docstrings

exception `ssh.SSHMultiplexingException`

Called when there's a failure trying to open a sub-shell via OpenSSH's [Master mode](#) multiplexing capability.

exception `ssh.SSHExecutionException`

Called when there's an error trying to execute a command in the slave.

exception `ssh.SSHKeygenException`

Called when there's an error trying to generate a public/private keypair.

exception `ssh.SSHKeypairException`

Called when there's an error trying to save public/private keypair or certificate.

exception `ssh.SSHPassphraseException`

Called when we try to generate/decode something that requires a passphrase but no passphrase was given.

`ssh.get_ssh_dir(self)`

Given a `gateone.TerminalWebSocket(self)` instance, return the current user's ssh directory

Note: If the user's ssh directory doesn't start with a `.` (dot) it will be renamed.

`ssh.open_sub_channel(self, term)`

Opens a sub-channel of communication by executing a new shell on the SSH server using OpenSSH's `Master mode` capability (it spawns a new slave) and returns the resulting `termio.Multiplex` instance. If a slave has already been opened for this purpose it will re-use the existing channel.

`ssh.wait_for_prompt(term, cmd, errorback, callback, m_instance, matched)`

Called by `termio.Multiplex.expect()` inside of `execute_command()`, clears the screen and executes `cmd`. Also, sets an `expect()` to call `get_cmd_output()` when the end of the command output is detected.

`ssh.get_cmd_output(term, errorback, callback, m_instance, matched)`

Captures the output of the command executed inside of `wait_for_prompt()` and calls `callback` if it isn't `None`.

`ssh.terminate_sub_channel(m_instance)`

Calls `m_instance.terminate()` and deletes it from `OPEN_SUBCHANNELS`.

`ssh.timeout_sub_channel(m_instance)`

Called when the sub-channel times out by way of an `termio.Multiplex.expect` pattern that should never match anything.

`ssh.got_error(self, m_instance, match=None, term=None, cmd=None)`

Called if `execute_command()` encounters a problem/timeout.

`match` is here in case we want to use it for a positive match of an error.

`ssh.execute_command(self, term, cmd, callback=None)`

Execute the given command (`cmd`) on the given `term` using the existing SSH tunnel (taking advantage of `Master mode`) and call `callback` with the output of said command and the current `termio.Multiplex` instance as arguments like so:

```
callback(output, m_instance)
```

If `callback` is not provided then the command will be executed and any output will be ignored.

Note: This will not result in a new terminal being opened on the client—it simply executes a command and returns the result using the existing SSH tunnel.

`ssh.send_result(self, term, cmd, output, m_instance)`

Called by `ws_exec_command()` when the output of the executed command has been captured successfully. Writes a message to the client with the command's output and some relevant metadata.

`ssh.ws_exec_command(self, settings)`

Takes the necessary variables from `settings` and calls `execute_command()`.

`settings` should be a dict that contains a 'term' and a 'cmd' to execute.

Tip: This function can be used to quickly execute a command and return its result from the client

over an existing SSH connection without requiring the user to enter their password! See `execRemoteCmd()` in `ssh.js`.

class `ssh.KnownHostsHandler(application, request, **kwargs)`

This handler allows the client to view, edit, and upload the `known_hosts` file associated with their user account.

get(**args, **kwargs*)

Determine what the user is asking for and call the appropriate method.

post(**args, **kwargs*)

Determine what the user is updating by checking the given arguments and proceed with the update.

_return_known_hosts()

Returns the user's `known_hosts` file in text/plain format.

_save_known_hosts(*known_hosts*)

Save the given *known_hosts* file.

`ssh.get_connect_string(self, term)`

Attached to the (server-side) `terminal:ssh_get_connect_string` WebSocket action; writes the connection string associated with *term* to the WebSocket like so:

```
{'terminal:sshjs_reconnect': {'term': <connection string>}}
```

In `ssh.js` we attach a WebSocket action to `'terminal:sshjs_reconnect'` that assigns the connection string sent by this function to `GateOne.Terminal.terminals[*term]['sshConnectString']`.

`ssh.get_key(self, name, public)`

Returns the private SSH key associated with *name* to the client. If *public* is `True`, returns the public key to the client.

`ssh.get_public_key(self, name)`

Returns the user's public key file named *name*.

`ssh.get_private_key(self, name)`

Returns the user's private key file named *name*.

`ssh.get_host_fingerprint(self, settings)`

Returns a the hash of the given host's public key by making a remote connection to the server (not just by looking at `known_hosts`).

`ssh.generate_new_keypair(self, settings)`

Calls `openssh_generate_new_keypair()` or `dropbear_generate_new_keypair()` depending on what's available on the system.

`ssh.overwrite(m_instance, match)`

Called if we get asked to overwrite an existing keypair.

`ssh.openssh_generate_new_keypair(self, name, path, keytype=None, passphrase="", bits=None, comment="")`

Generates a new private and public key pair—stored in the user's directory using the given *name* and other optional parameters (using OpenSSH).

If *keytype* is given, it must be one of `"ecdsa"`, `"rsa"` or `"dsa"` (case insensitive). If *keytype* is `"rsa"` or `"ecdsa"`, *bits* may be specified to specify the size of the key.

Note: Defaults to generating a 521-byte `ecdsa` key if OpenSSH is version 5.7+. Otherwise a 2048-bit `rsa` key will be used.

`ssh.dropbear_generate_new_keypair(self, name, path, keytype=None, passphrase="", bits=None, comment="")`

Note: Not implemented yet

`ssh.openssh_generate_public_key(self, path, passphrase=None, settings=None)`

Generates a public key from the given private key at *path*. If a *passphrase* is provided, it will be used to generate the public key (if necessary).

`ssh.store_id_file(self, settings)`

Stores the given *settings['private']* and/or *settings['public']* keypair in the user's ssh directory as *settings['name']* and/or *settings['name'].pub*, respectively. Either file can be saved independent of each other (in case this function needs to be called multiple times to save each respective file).

Also, a *settings['certificate']* may be provided to be saved along with the private and public keys. It will be saved as *settings['name']-cert.pub*.

Note: I've found the following website helpful in understanding how to use OpenSSH with SSL certificates: <http://blog.habets.pp.se/2011/07/OpenSSH-certificates>

Tip: Using signed-by-a-CA certificates is very handy because allows you to revoke the user's SSH key(s). e.g. If they left the company.

`ssh.delete_identity(self, name)`

Removes the identity associated with *name*. For example if *name* is 'testkey', 'testkey' and 'testkey.pub' would be removed from the user's ssh directory (and 'testkey-cert.pub' if present).

`ssh.get_identities(self, *anything)`

Sends a message to the client with a list of the identities stored on the server for the current user. *anything* is just there because the client needs to send *something* along with the 'action'.

`ssh.set_default_identities(self, identities)`

Given a list of *identities*, mark them as defaults to use in all outbound SSH connections by writing them to <user's ssh dir>/.default_ids. If *identities* is empty, no identities will be used in outbound connections.

Note: Whenever this function is called it will overwrite whatever is in *default_ids*.

`ssh.set_ssh_socket(self, term, path)`

Given a *term* and *path*, sets *self.loc_terms[term]['ssh_socket']* = *path*.

`ssh.set_ssh_connect_string(self, term, connect_string)`

Given a *term* and *connect_string*, sets *self.loc_terms[term]['ssh_connect_string']* = *connect_string*.

`ssh.opt_esc_handler(self, text, term=None, multiplex=None)`

Handles text passed from the special optional escape sequence handler. We use it to tell *ssh.js* what the SSH connection string is so it can use that information to duplicate sessions (if the user so desires). For reference, the specific string which will call this function from a terminal app is:

```
\x1b_ ;ssh|<whatever>\x07
```

See also:

`gateone.TerminalWebSocket.opt_esc_handler` and `terminal.Terminal._opt_handler()`

`ssh.create_user_ssh_dir(self)`

To be called by the 'Auth' hook that gets called after the user is done authenticating, ensures that the `<user's dir>/ssh` directory exists.

`ssh.send_ssh_css_template(self)`

Sends our `ssh.css` template to the client using the 'load_style' WebSocket action. The rendered template will be saved in Gate One's 'cache_dir'.

`ssh.initialize(self)`

Called inside of `TerminalApplication.initialize()` shortly after the WebSocket is instantiated. Attaches our two `terminal:authenticate` events (to create the user's `.ssh` dir and send our CSS template) and ensures that the `ssh_connect.py` script is executable.

1.3 Developer Documentation

1.3.1 Python Code

Gate One consists of `gateone.py` and several supporting Python modules and scripts. The documentation for each can be found below:

`authentication.py` - Authentication Classes

Authentication

This module contains Gate One's authentication classes. They map to Gate One's `--auth` configuration option like so:

<code>--auth=None</code>	<code>NullAuthHandler</code>
<code>--auth=kerberos</code>	<code>KerberosAuthHandler</code>
<code>--auth=google</code>	<code>GoogleAuthHandler</code>
<code>--auth=pam</code>	<code>PAMAuthHandler</code>
<code>--auth=api</code>	<code>APIAuthHandler</code>

Note: API authentication is handled inside of *Gate One*

None or Anonymous By default Gate One will not authenticate users. This means that user sessions will be tied to their browser cookie and users will not be able to resume their sessions from another computer/browser. Most useful for situations where session persistence and logging aren't important.

All users will show up as ANONYMOUS using this authentication type.

Kerberos Kerberos authentication utilizes GSSAPI for Single Sign-on (SSO) but will fall back to HTTP Basic authentication if GSSAPI auth fails. This authentication type can be integrated into any Kerberos infrastructure including Windows Active Directory.

It is great for both transparent authentication and being able to tie sessions and logs to specific users within your organization (compliance).

Note: The `sso.py` module itself has extensive documentation on this authentication type.

Google Authentication If you want persistent user sessions but don't care to run your own authentication infrastructure this authentication type is for you. Assuming, of course, that your Gate One server and clients will have access to the Internet.

Note: This authentication type is perfect if you're using Chromebooks (Chrome OS devices).

API Authentication API-based authentication is actually handled in `gateone.py` but we still need *something* to exist at the `/auth` URL that will always return the 'unauthenticated' response. This ensures that no one can authenticate themselves by visiting that URL manually.

Docstrings

`gateone.auth.authentication.additional_attributes(user, settings_dir=None)`

Given a *user* dict, return a dict containing any additional attributes defined in Gate One's attribute repositories.

Note: This function doesn't actually work yet (support for attribute repos like LDAP is forthcoming).

class `gateone.auth.authentication.BaseAuthHandler(application, request, **kwargs)`

The base class for all Gate One authentication handlers.

get_current_user()

Tornado standard method—implemented our way.

user_login(user)

Called immediately after a user authenticates successfully. Saves session information in the user's directory. Expects *user* to be a dict containing a 'upn' value representing the username or userPrincipalName. e.g. 'user@REALM' or just 'someuser'. Any additional values will be attached to the user object/cookie.

user_logout(user, redirect=None)

Called immediately after a user logs out, cleans up the user's session information and optionally, redirects them to *redirect* (URL).

class `gateone.auth.authentication.NullAuthHandler(application, request, **kwargs)`

A handler for when no authentication method is chosen (i.e. `-auth=none`). With this handler all users will show up as "ANONYMOUS".

get(*args, **kwargs)

Sets the 'gateone_user' cookie with a new random session ID (*go_session*) and sets *go_upn* to 'ANONYMOUS'.

user_login(user)

This is an override of `BaseAuthHandler` since anonymous auth is special. Generates a unique session ID for this user and saves it in a browser cookie. This is to ensure that anonymous users can't access each other's sessions.

class `gateone.auth.authentication.APIAuthHandler(application, request, **kwargs)`

A handler that always reports 'unauthenticated' since API-based auth doesn't use auth handlers.

get(*args, **kwargs)

Deletes the 'gateone_user' cookie and handles some other situations for backwards compatibility.

class `gateone.auth.authentication.GoogleAuthHandler(application, request, **kwargs)`

Google authentication handler using Tornado's built-in `GoogleOAuth2Mixin` (fairly boilerplate).

get(*args, **kwargs)
Sets the 'user' cookie with an appropriate *upn* and *session* and any other values that might be attached to the user object given to us by Google.

_on_auth(user)
Just a continuation of the `get()` method (the final step where it actually sets the cookie).

class `gateone.auth.authentication.SSLAuthHandler(application, request, **kwargs)`
SSL Certificate-based authentication handler. Can only be used if the `ca_certs` option is set along with `ssl_auth=required` or `ssl_auth=optional`.

initialize()
Print out helpful error messages if the requisite settings aren't configured.

_convert_certificate(cert)
Converts the certificate format returned by `get_ssl_certificate()` into a format more suitable for a user dict.

get(*args, **kwargs)
Sets the 'user' cookie with an appropriate *upn* and *session* and any other values that might be attached to the user's client SSL certificate.

class `gateone.auth.authentication.KerberosAuthHandler(application, request, **kwargs)`
Handles authenticating users via Kerberos/GSSAPI/SSO.

get(*args, **kwargs)
Checks the user's request header for the proper Authorization data. If it checks out the user will be logged in via `_on_auth()`. If not, the browser will be redirected to login.

class `gateone.auth.authentication.PAMAuthHandler(application, request, **kwargs)`
Handles authenticating users via PAM.

get(*args, **kwargs)
Checks the user's request header for the proper Authorization data. If it checks out the user will be logged in via `_on_auth()`. If not, the browser will be redirected to login.

class `gateone.auth.authentication.CASAuthHandler(application, request, **kwargs)`
CAS authentication handler.

initialize()
Print out helpful error messages if the requisite settings aren't configured.

NOTE: It won't hurt anything to override this method in your RequestHandler.

get(*args, **kwargs)
Sets the 'user' cookie with an appropriate *upn* and *session* and any other values that might be attached to the user object given to us by Google.

authenticate_redirect(callback=None)
Redirects to the authentication URL for this CAS service.

After authentication, the service will redirect back to the given callback URI with additional parameters.

We request the given attributes for the authenticated user by default (name, email, language, and username). If you don't need all those attributes for your app, you can request fewer with the `ax_attrs` keyword argument.

get_authenticated_user(server_ticket)
Requests the user's information from the CAS server using the given *server_ticket* and calls `self._on_auth` with the resulting user dict.

`_on_auth(response)`

Just a continuation of the `get()` method (the final step where it actually sets the cookie).

authorization.py - Authentication Classes

Authorization

This module contains Gate One's authorization helpers.

Docstrings

class `gateone.auth.authorization.require(*conditions)`

A decorator to add authorization requirements to any given function or method using condition classes. Condition classes are classes with `check()` methods that return `True` if the condition is met.

Example of using `@require` with `is_user()`:

```
@require(is_user('administrator'))
def admin_index(self):
    return 'Hello, Administrator!'
```

This would only allow the user, 'administrator' access to the index page. In this example the *condition* is the `is_user` function which checks that the logged-in user's username (aka UPN) is 'administrator'.

class `gateone.auth.authorization.authenticated`

A condition class to be used with the `@require` decorator that returns `True` if the user is authenticated.

Note: Only meant to be used with WebSockets. `tornado.web.RequestHandler` instances can use `@tornado.web.authenticated`

class `gateone.auth.authorization.is_user(upn)`

A condition class to be used with the `@require` decorator that returns `True` if the given username/UPN matches what's in `self._current_user`.

class `gateone.auth.authorization.policies(app)`

A condition class to be used with the `@require` decorator that returns `True` if all the given conditions are within the limits specified in Gate One's settings (e.g. `50limits.conf`). Here's an example:

```
@require(authenticated(), policies('terminal'))
def new_terminal(self, settings):
    # Actual function would be here
    pass
```

That would apply all policies that are configured for the 'terminal' application. It works like this:

The `TerminalApplication` application registers its name and policy-checking function inside of `initialize()` like so:

```
self.ws.security.update({'terminal': terminal_policies})
```

Whenever a function decorated with `@require(policies('terminal'))` is called the registered policy-checking function (e.g. `app_terminal.terminal_policies()`) will be called, passing the current instance of `policies` as the only argument.

It is then up to the policy-checking function to make a determination as to whether or not the user is allowed to execute the decorated function and must return `True` if allowed. Also note that the

policy-checking function will be able to make modifications to the function and its arguments if the security policies warrant it.

Note: If you write your own policy-checking function (like `terminal_policies()`) it is often a good idea to send a notification to the user indicating why they've been denied. You can do this with the `instance.send_message()` method.

ctypes_pam.py - PAM Authentication Module

PAM Authentication Module for Python

Provides an `authenticate` function that will allow the caller to authenticate a user against the Pluggable Authentication Modules (PAM) on the system.

Implemented using `ctypes`, so no compilation is necessary.

```
gateone.auth.ctypes_pam.authenticate(username, password, service='login', tty='console',
                                     **kwargs)
```

Returns True if the given username and password authenticate for the given service. Returns False otherwise.

Parameters

- **username** (*string*) – The username to authenticate.
- **password** (*string*) – The password in plain text.
- **service** (*string*) – The PAM service to authenticate against. Defaults to 'login'.
- **tty** (*string*) – Name of the TTY device to use when authenticating. Defaults to 'console' (to allow root).

If additional keyword arguments are provided they will be passed to `PAM_SET_ITEM()` like so:

```
PAM_SET_ITEM(handle, <keyword mapped to PAM_whatever>, <value>)
```

Where the keyword will be automatically converted to a `PAM_whatever` constant if present in this file. Example:

```
authenticate(user, pass, PAM_RHOST="myhost")
```

...would result in:

```
PAM_SET_ITEM(handle, 4, "myhost") # PAM_RHOST (4) taken from the global
```

pam.py - A PAM Authentication Module

PAM Authentication Module for Gate One

This authentication module is built on top of *ctypes_pam.py - PAM Authentication Module* which is included with Gate One.

It was originally written by Alan Schmitz (but has changed quite a bit).

The only non-obvious aspect of this module is that the `pam_realm` setting is only used when the user is asked to authenticate and when the user's information is stored in the 'users' directory. It isn't actually used in any part of the authentication (PAM doesn't take a "realm" setting).

class gateone.auth.pam.PAMAuthMixin(*application, request, **kwargs*)

This is used by PAMAuthHandler in *Authentication* to authenticate users via PAM.

initialize()

Print out helpful error messages if the requisite settings aren't configured.

get_authenticated_user(*callback*)

Processes the client's Authorization header and call self.auth_basic()

auth_basic(*auth_header, callback*)

Perform Basic authentication using self.settings['pam_realm'].

authenticate_redirect()

Informs the browser that this resource requires authentication (status code 401) which should prompt the browser to reply with credentials.

The browser will be informed that we support Basic auth.

sso.py - A Tornado Kerberos Single Sign-On Module

About The SSO Module

sso.py is a Tornado Single Sign-On (SSO) authentication module that implements GSSAPI authentication via python-kerberos (import kerberos). If "Negotiate" authentication (GSSAPI SSO) fails it will gracefully fall back to "Basic" auth (authenticating a given username/password against your Kerberos realm).

For this module to work you must add 'sso_realm' and 'sso_service' to your Tornado application's settings. See the docstring of the KerberosAuthMixin for how to do this.

This module should work with regular MIT Kerberos implementations as well as Active Directory (Heimdal is untested but should work fine). If you're experiencing trouble it is recommended that you set debug=True in your application settings. This will enable printing of Kerberos exception messages.

Troubleshooting If your browser asks you for a password (i.e. SSO failed) there's probably something wrong with your Kerberos configuration on either the client or the server (usually it's a problem with forward/reverse DNS resolution or an incorrect or missing service principal in your keytab).

If you're using Active Directory, make sure that there's an HTTP servicePrincipalName (SPN) matching the FQDN of the host running your Tornado server. For example: [HTTP/somehost.somedomain.com@CORP.MYCOMPANY.COM](#) You may also want a short hostname SPN: [HTTP/somehost@CORP.MYCOMPANY.COM](#)

Also make sure that the service principal is in upper case as most clients (web browsers) will auto-capitalise the principal when verifying the server.

Here's some things to test in order to find problems with your Kerberos config:

Try these from both the client and the server (NOTE: Assuming both are Unix): `kinit -p <user@REALM>`
To verify you can authenticate via Kerberos (at all) `nslookup <server FQDN>` # To verify the IP address reverse maps properly (below) `nslookup <IP address that 'server FQDN' resolves to> kvno HTTP/somehost.somedomain.com` # To verify your service principal

Remember: Kerberos is heavily dependent on DNS to verify the server and client are who they claim to be.

I find that it is useful to get GSSAPI authentication working with OpenSSH first before I attempt to get a custom service principal working with other applications. This is because SSH uses the HOST/ principal which is often taken care of automatically via most Kerberos management tools (including AD). If you can get SSO working with SSH you can get SSO working with anything else.

Class Docstrings

class gateone.auth.sso.KerberosAuthMixin(application, request, **kwargs)

Authenticates users via Kerberos-based Single Sign-On. Requires that you define 'sso_realm' and 'sso_service' in your Tornado Application settings. For example:

```
settings = dict(
    cookie_secret="iYR123qg4Udsgf4CRung6BFUBhizAciid8oq1YfJR3gN",
    static_path=os.path.join(os.path.dirname(__file__), "static"),
    gzip=True,
    login_url="/auth",
    debug=True,
    sso_realm="EXAMPLE.COM",
    sso_service="HTTP" # Should pretty much always be HTTP
)
```

NOTE: If you're using 'HTTP' as the service it must be in all caps or it might not work with some browsers/clients (which auto-capitalizes all services).

To implement this mixin:

```
from sso import KerberosAuthMixin
class KerberosAuthHandler(tornado.web.RequestHandler, KerberosAuthMixin):

    def get(self):
        auth_header = self.request.headers.get('Authorization')
        if auth_header:
            self.get_authenticated_user(self._on_auth)
            return
        self.authenticate_redirect()

    def _on_auth(self, user):
        if not user:
            raise tornado.web.HTTPError(500, "Kerberos auth failed")
        self.set_secure_cookie("user", tornado.escape.json_encode(user))
        print("KerberosAuthHandler user: %s" % user) # To see what you get
        next_url = self.get_argument("next", None) # To redirect properly
        if next_url:
            self.redirect(next_url)
        else:
            self.redirect("/")
```

initialize()

Print out helpful error messages if the requisite settings aren't configured.

NOTE: It won't hurt anything to override this method in your RequestHandler.

get_authenticated_user(callback)

Processes the client's Authorization header and calls self.auth_negotiate() or self.auth_basic() depending on what headers were provided by the client.

auth_negotiate(auth_header, callback)

Perform Negotiate (GSSAPI/SSO) authentication via Kerberos.

auth_basic(auth_header, callback)

Perform Basic authentication using Kerberos against self.settings['sso_realm'].

authenticate_redirect()

Informs the browser that this resource requires authentication (status code 401) which should prompt the browser to reply with credentials.

The browser will be informed that we support both Negotiate (GSSAPI/SSO) and Basic auth.

log.py - Gate One Logging Module

Logging Module for Gate One

This module contains a number of pre-defined loggers for use within Gate One:

Name	Description
go_log	Used for logging internal Gate One events.
auth_log	Used for logging authentication and authorization events.
msg_log	Used for logging messages sent to/from users.

Applications may also use their own loggers for differentiation purposes. Such loggers should be prefixed with 'gateone.app.' like so:

```
>>> import logging
>>> logger = logging.getLogger("gateone.app.myapp")
```

Additional loggers may be defined within a GOApplication with additional prefixing:

```
>>> xfer_log = logging.getLogger("gateone.app.myapp.xfer")
>>> lookup_log = logging.getLogger("gateone.app.myapp.lookup")
```

Note: This module does not cover session logging within the Terminal application. That is a built-in feature of the `termio` module.

exception gateone.core.log.UnknownFacility

Raised if `string_to_syslog_facility` is given a string that doesn't match a known syslog facility.

class gateone.core.log.JSONAdapter(*logger, extra*)

A `logging.LoggerAdapter` that prepends keyword argument information to log entries. Expects the passed in dict-like object which will be included.

Initialize the adapter with a logger and a dict-like object which provides contextual information. This constructor signature allows easy stacking of `LoggerAdapters`, if so desired.

You can effectively pass keyword arguments as shown in the following example:

```
adapter = LoggerAdapter(someLogger, dict(p1=v1, p2="v2"))
```

gateone.core.log.string_to_syslog_facility(*facility*)

Given a string (*facility*) such as, "daemon" returns the numeric syslog.LOG_* equivalent.

gateone.core.log.go_logger(*name, **kwargs*)

Returns a new `logging.Logger` instance using the given *name* pre-configured to match Gate One's usual logging output. The given *name* will automatically be prefixed with 'gateone.' if it is not already. So if *name* is 'app.foo' the Logger would end up named 'gateone.app.foo'. If the given *name* is already prefixed with 'gateone.' it will be left as-is.

The log will be saved in the same location as Gate One's configured `log_file_prefix` using the given *name* with the following convention:

```
gateone/logs/<modified *name*>.log
```

The file name will be modified like so:

- It will have the 'gateone' portion removed (since it's redundant).
- Dots will be replaced with dashes (-).

Examples:

```
>>> auth_logger = go_logger('gateone.auth.terminal')
>>> auth_logger.info('test1')
>>> app_logger = go_logger('gateone.app.terminal')
>>> app_logger.info('test2')
>>> import os
>>> os.listdir('/opt/gateone/logs')
['auth.log', 'auth-terminal.log', 'app-terminal.log', 'webserver.log']
```

If any *kwargs* are given they will be JSON-encoded and included in the log message after the date/metadata like so:

```
>>> auth_logger.info('test3', {"user": "bob", "ip": "10.1.1.100"})
[I 130828 15:00:56 app.py:10] {"user": "bob", "ip": "10.1.1.100"} test3
```

logviewer.py - Session Log Viewer

Log Viewer

Allows the user to play back a given log file like a video (default) or display it in a syslog-like format. To view usage information, run it with the `--help` switch:

```
root@host:/opt/gateone $ ./logviewer.py --help
Usage: logviewer.py [options] <log file>
```

Options:

```
--version      show program's version number and exit
-h, --help     show this help message and exit
-f, --flat     Display the log line-by-line in a syslog-like format.
-p, --playback Play back the log in a video-like fashion. This is the
               default view.
--pretty       Preserve font and character renditions when displaying the
               log in flat view (default).
--raw          Display control characters and escape sequences when
               viewing.
```

Here's an example of how to display a Gate One log (.golog) in a flat, greppable format:

```
root@host:/opt/gateone $ ./logviewer.py --flat
Sep 09 21:07:14 Host/IP or SSH URL [localhost]: modern-host
Sep 09 21:07:16 Port [22]:
Sep 09 21:07:16 User: bsmith
Sep 09 21:07:17 Connecting to: ssh://bsmith@modern-host:22
Sep 09 21:07:17
Sep 09 21:07:17 bsmith@modern-host's password:
Sep 09 21:07:20 Welcome to Ubuntu 11.04 (GNU/Linux 2.6.38-11-generic x86_64)
Sep 09 21:07:20
Sep 09 21:07:20 * Documentation: https://help.ubuntu.com/
Sep 09 21:07:20
Sep 09 21:07:20 Last login: Thu Sep 29 08:51:27 2011 from portarisk
Sep 09 21:07:20 bsmith@modern-host:~ $ ls
Sep 09 21:07:21 why_I_love_gate_one.txt  to_dont_list.txt
Sep 09 21:07:21 bsmith@modern-host:~ $
```

About Gate One's Log Format

Gate One's log format (.golog) is a gzip-compressed unicode (UTF-8) text file consisting of time-based frames separated by the unicode character, U+F0F0F0. Each frame consists of JavaScript-style timestamp (because it is compact) followed by a colon and then the text characters of the frame. A frame ends when a U+F0F0F0 character is encountered.

Here are two example .golog frames demonstrating the format:

```
1317344834868:\x1b[H\x1b[2JHost/IP or SSH URL [localhost]: <U+F0F0F>1317344836086:\r\nPort [22]: <U+F0F0F>
```

Gate One logs can be opened, decoded, and parsed in Python fairly easily:

```
import gzip
golog = gzip.open(path_to_golog).read()
for frame in golog.split(u"\U000f0f0f".encode('UTF-8')):
    frame_time = float(frame[:13]) # First 13 chars is the timestamp
    # Timestamps can be converted into datetime objects very simply:
    datetime_obj = datetime.fromtimestamp(frame_time/1000)
    frame_text = frame[14:] # This gets you the actual text minus the colon
    # Do something with the datetime_obj and the frame_text
```

Note: U+F0F0F0 is from Private Use Area (PUA) 15 in the Unicode Character Set (UCS). It was chosen at random (mostly =) from PUA-15 because it is highly unlikely to be used in an actual terminal program where it could corrupt a session log.

Class Docstrings

`logviewer.get_frames(golog_path, chunk_size=131072)`

A generator that iterates over the frames in a .golog file, returning them as strings.

`logviewer.retrieve_first_frame(golog_path)`

Retrieves the first frame from the given *golog_path*.

`logviewer.get_log_metadata(golog_path)`

Returns the metadata from the log at the given *golog_path* in the form of a dict.

`logviewer.playback_log(log_path, file_like, show_esc=False)`

Plays back the log file at *log_path* by way of timely output to *file_like* which is expected to be any file-like object with `write()` and `flush()` methods.

If *show_esc* is True, escape sequences and control characters will be escaped so they can be seen in the output. There will also be no delay between the output of frames (under the assumption that if you want to see the raw log you want it to output all at once so you can pipe it into some other app).

`logviewer.escape_escape_seq(text, preserve_renditions=True, rstrip=True)`

Escapes escape sequences so they don't muck with the terminal viewing *text* Also replaces special characters with unicode symbol equivalents (e.g. so you can see what they are without having them do anything to your running shell)

If *preserve_renditions* is True, CSI escape sequences for renditions will be preserved as-is (e.g. font color, background, etc).

If *rstrip* is true, trailing escape sequences and whitespace will be removed.

`logviewer.flatten_log(log_path, file_like, preserve_renditions=True, show_esc=False)`

Given a log file at *log_path*, write a string of log lines contained within to *file_like*. Where *file_like* is

expected to be any file-like object with `write()` and `flush()` methods.

If `preserve_renditions` is `True`, CSI escape sequences for renditions will be preserved as-is (e.g. font color, background, etc). This is to make the output appear as close to how it was originally displayed as possible. Besides that, it looks really nice =)

If `show_esc` is `True`, escape sequences and control characters will be visible in the output. Trailing whitespace and escape sequences will not be removed.

..note:

Converts our standard recording-based log format into something that can be used with `grep` and similar search/filter tools.

`logviewer.render_log_frames(golog_path, rows, cols, limit=None)`

Returns the frames of `golog_path` as a list of HTML-encoded strings that can be used with the `playback_log.html` template. It accomplishes this task by running the frames through the terminal emulator and capturing the HTML output from the `Terminal.dump_html` method.

If `limit` is given, only return that number of frames (e.g. for preview)

`logviewer.get_256_colors(container='gateone')`

Returns the rendered 256-color CSS. If `container` is provided it will be used as the `{{container}}` variable when rendering the template (defaults to "gateone").

`logviewer.render_html_playback(golog_path, render_settings=None)`

Generates a self-contained HTML playback file from the `.golog` at the given `golog_path`. The HTML will be output to `stdout`. The optional `render_settings` argument (dict) can include the following options to control how the output is rendered:

prefix (Default: "go_default_") The `GateOne.prefs.prefix` to emulate when rendering the HTML template.

container (Default: "gateone") The name of the `#gateone` container to emulate when rendering the HTML template.

theme (Default: "black") The theme to use when rendering the HTML template.

colors (Default: "default") The text color scheme to use when rendering the HTML template.

Note: This function returns a byte string (not a unicode string).

`logviewer.get_terminal_size()`

Returns the size of the current terminal in the form of (rows, cols).

`logviewer.main(args=['usr/local/bin/sphinx-build', '-b', 'html', '-v', '-T', '-d', 'build/doctrees', 'source', 'build/html'])`

Parse command line arguments and view the log in the specified format.

server.py - Gate One's Core Script

Gate One

Gate One is a web-based terminal emulator written in Python using the Tornado web framework. This module runs the primary daemon process and acts as a central controller for all running terminals and terminal programs. It supports numerous configuration options and can also be called with the `--kill` switch to kill all running terminal programs (if using `dtach`—otherwise they die on their own when `gateone.py` is stopped).

Dependencies Gate One requires Python 2.6+ but runs best with Python 2.7+. It also depends on the following 3rd party Python modules:

- **Tornado** 3.1+ - A non-blocking web server framework that powers FriendFeed.

The following modules are optional and can provide Gate One with additional functionality:

- **pyOpenSSL** 0.10+ - An OpenSSL module/wrapper for Python. Only used to generate self-signed SSL keys and certificates. If pyOpenSSL isn't available Gate One will fall back to using the 'openssl' command to generate self-signed certificates.
- **kerberos** 1.0+ - A high-level Kerberos interface for Python. Only necessary if you plan to use the Kerberos authentication module.
- **python-pam** 0.4.2+ - A Python module for interacting with PAM (the Pluggable Authentication Module present on nearly every Unix). Only necessary if you plan to use PAM authentication.
- **PIL (Python Imaging Library)** 1.1.7+ - A Python module for manipulating images. **Alternative:** **Pillow** 2.0+ - A "friendly fork" of PIL that works with Python 3.
- **mutagen** 1.21+ - A Python module to handle audio metadata. Makes it so that if you cat `music_file.ogg` in a terminal you'll get useful track/tag information.

With the exception of python-pam, all required and optional modules can usually be installed via one of these commands:

```
user@modern-host:~ $ sudo pip install --upgrade tornado pyopenssl kerberos pillow mutagen
```

...or:

```
user@legacy-host:~ $ sudo easy_install tornado pyopenssl kerberos pillow mutagen
```

Note: The use of pip is recommended. See <http://www.pip-installer.org/en/latest/installing.html> if you don't have it.

The python-pam module is available in most Linux distribution repositories. Simply executing one of the following should take care of it:

```
user@debian-or-ubuntu-host:~ $ sudo apt-get install python-pam
```

```
user@redhat-host:~ $ sudo yum install python-pam
```

```
user@gentoo-host:~ $ sudo emerge python-pam
```

```
user@suse-host:~ $ sudo yast -i python-pam
```

Settings Most of Gate One's options can be controlled by command line switches or via .conf files in the settings directory. If you haven't configured Gate One before a number of .conf files will be automatically generated using defaults and/or the command line switches provided the first time you run `gateone.py`.

Settings in the various settings/*.conf files are JSON-formatted:

Note: Technically, JSON doesn't allow comments but Gate One's .conf files do.

```
{ // You can use single-line comments like this
/*
Or multi-line comments like this.
*/
```

```

"*": { // The * here designates this as "default" values
  "gateone": { // Settings in this dict are specific to the "gateone" app
    "address": "10.0.0.100", // A string value
    "log_file_num_backups": 10, // An integer value
    // Here's an example list:
    "origins": ["localhost", "127.0.0.1", "10.0.0.100"],
    "https_redirect": false, // Booleans are all lower case
    "log_to_stderr": null // Same as `None` in Python (also lower case)
    // NOTE: No comma after last item
  },
  "terminal": { // Example of a different application's settings
    "default_command": "SSH", // <-- don't leave trailing commas like this!
    ... // So on and so forth
  }
}
}

```

Note: You *must* use double quotes ("") to define strings. Single quotes *can* be used inside double quotes, however. "example": "of 'single' quotes" To escape a double-quote inside a double quote use three slashes: "\\\"foo\\\""

You can have as many .conf files in your settings directory as you like. When Gate One runs it reads all files in alphanumeric order and combines all settings into a single Python dictionary. Files loaded last will override settings from earlier files. Example:

20example.conf

```

{
  "*": {
    "terminal": {
      "session_logging": true,
      "default_command": "SSH"
    }
  }
}

```

99override.conf

```

{
  "*": {
    "terminal": {
      "default_command": "my_override"
    }
  }
}

```

If Gate One loaded the above example .conf files the resulting dict would be:

Merged settings

```
{
  "*": {
    "terminal": {
      "session_logging": true,
      "default_command": "my_override"
    }
  }
}
```

Note: These settings are loaded using the RUDict (Recursive Update Dict) class.

There are a few important differences between the configuration file and command line switches in regards to boolean values (True/False). A switch such as `--debug` is equivalent to `"debug" = true` in `10server.conf`:

```
"debug" = true // Booleans are all lower case (not in quotes)
```

Tip: Use `--setting=True`, `--setting=False`, or `--setting=None` to avoid confusion.

Note: The following values in `10server.conf` are case sensitive: `true`, `false` and `null` (and should not be placed in quotes).

Gate One's configuration files also provide access control mechanisms. These are controlled by setting the *scope* of the configuration block. In this example all users that match the given IP address will be denied access to Gate One:

```
{
  "user.ip_address=(127.0.0.1|10.1.1.100)": { // Replaces "*"
    "gateone": { // These settings apply to all of Gate One
      "blacklist": true,
    }
  }
}
```

You can define scopes however you like using user's attributes. For example:

```
{
  "user.email=.*@company.com": {
    "terminal": {
      "commands": {"extra": "/some/extra/command.sh"}
    }
  }
}
```

The above example would make it so that all users with an email address ending in `'@company.com'` will get access to the "extra" command when using the "terminal" application.

Note: Different authentication types provide different user attributes. For example, if you have "auth" set to "google" authenticated users will have a 'gender' attribute. So for example—if you wanted to be evil—you could provide different settings for men and women (e.g. `"user.gender=male"`).

Tip: When using API authentication you can pass whatever extra user attributes you want via the 'auth'

object. These attributes will be automatically assigned to the user and can be used with the policy mechanism to control access and settings.

Running `gateone.py` with the `--help` switch will print the usage information as well as descriptions of what each configurable option does:

```
$ gateone --help
[W 150205 13:58:35 utils:940] Could not import entry point module: gateone.applications.x11
Usage: /usr/local/bin/gateone [OPTIONS]
```

Options:

```
--help                Show this help information
```

`/usr/local/lib/python2.7/dist-packages/tornado/log.py` options:

```
--log_file_max_size    max size of log files before rollover (default 100000000)
--log_file_num_backups  number of log files to keep (default 10)
--log_file_prefix=PATH  Path prefix for log files. Note that if you are running multiple tornado processes, log
                        port number)
--log_to_stderr         Send log output to stderr (colorized if possible). By default use stderr if --log_file
--logging=debug|info|warning|error|none
                        Set the Python log level. If 'none', tornado won't touch the logging configuration. (default info)
```

gateone options:

```
--address              Run on the given address. Default is all addresses (IPv6 included). Multiple addresses
                        '127.0.0.1;:::1;10.1.1.100').
--api_keys              The 'key:secret,...' API key pairs you wish to use (only applies if using API authentication)
--api_timestamp_window  How long before an API authentication object becomes invalid. (default 30s)
--auth                 Authentication method to use. Valid options are: none, api, cas, google, ssl, kerberos, etc.
--ca_certs              Path to a file containing any number of concatenated CA certificates in PEM format. If not
                        set to 'optional' or 'required'.
--cache_dir             Path where Gate One should store temporary global files (e.g. rendered templates, CSS, etc.)
--certificate           Path to the SSL certificate. Will be auto-generated if not found. (default /etc/gateone/certificate.pem)
--combine_css           Combines all of Gate One's CSS Template files into one big file and saves it to the given path
--combine_css_container Use this setting in conjunction with --combine_css if the <div> where Gate One lives is in a container
--combine_js            Combines all of Gate One's JavaScript files into one big file and saves it to the given path
--command              DEPRECATED: Use the 'commands' option in the terminal settings.
--config               DEPRECATED. Use --settings_dir. (default /opt/gateone/server.conf)
--configure            Only configure Gate One (create SSL certs, conf.d, etc). Do not start any Gate One processes
--cookie_secret         Use the given 45-character string for cookie encryption.
--debug               Enable debugging features such as auto-restarting when files are modified. (default False)
--disable_ssl          If enabled Gate One will run without SSL (generally not a good idea). (default False)
--embedded            When embedding Gate One this option is available to plugins, applications, and templates.
--enable_unix_socket   Enable Unix socket support. (default False)
--gid                 Drop privileges and run Gate One as this group/gid. (default 1000)
--https_redirect       If enabled a separate listener will be started on port 80 that redirects users to the https listener
--js_init              A JavaScript object (string) that will be used when running GateOne.init() inside index.html
--keyfile              Path to the SSL keyfile. Will be auto-generated if none is provided. (default /etc/gateone/keyfile.pem)
--locale               The locale (e.g. pt_PT) Gate One should use for translations. If not provided, will default to en_US
--multiprocessing_workers The number of processes to spawn use when using multiprocessing. Default is: <number of processors>
--new_api_key          Generate a new API key that an external application can use to embed Gate One. (default None)
--origins              A semicolon-separated list of origins you wish to allow access to your Gate One server (e.g. http://foo;foo.bar;) and IP addresses. This value must contain all the hostnames/IPs that use
```

	specified to allow access from anywhere. NOTE: Using a '*' is only a good idea if you are running Gate One on localhost;127.0.0.1;enterprise;127.0.1.1)
--pam_realm	Basic auth REALM to display when authenticating clients. Default: hostname. Only relevant if PAM authentication is enabled.
--pam_service	PAM service to use. Defaults to 'login'. Only relevant if PAM authentication is enabled.
--pid_file	Define the path to the pid file. (default /home/riskable/.gateone/gateone.pid)
--port	Run on the given port. (default 10443)
--session_dir	Path to the location where session information will be stored. (default /home/riskable/.gateone/sessions)
--session_timeout	Amount of time that a session is allowed to idle before it is killed. Accepts <num>X where X is minutes, hours, or days. Set to '0' to disable the ability to resume sessions. (default 5d)
--settings_dir	Path to the settings directory. (default /home/riskable/.gateone/conf.d)
--ssl_auth	Enable the use of client SSL (X.509) certificates as a secondary authentication factor. Can be 'none', 'optional', or 'required'. NOTE: Only works if the 'ca_certs' option is configured.
--sso_realm	Kerberos REALM (aka DOMAIN) to use when authenticating clients. Only relevant if Kerberos authentication is enabled.
--sso_service	Kerberos service (aka application) to use. Only relevant if Kerberos authentication is enabled.
--syslog_facility	Syslog facility to use when logging to syslog (if syslog_session_logging is enabled). Can be local3, local4, local5, local6, local7, lpr, mail, news, syslog, user, uucp. (default local3)
--uid	Drop privileges and run Gate One as this user/uid. (default 1000)
--unix_socket_path	Path to the Unix socket (if --enable_unix_socket=True). (default /tmp/gateone.sock)
--url_prefix	An optional prefix to place before all Gate One URLs. e.g. '/gateone/'. Use this if Gate One is not located at some sub-URL path. (default /)
--user_dir	Path to the location where user files will be stored. (default /home/riskable/.gateone/users)
--user_logs_max_age	Maximum length of time to keep any given user log before it is automatically removed. (default 30d)
--version	Display version information.

terminal options:

--detach	Wrap terminals with dtach. Allows sessions to be resumed even if Gate One is stopped and restarted.
--kill	Kill any running Gate One terminal processes including dtach'd processes. (default False)
--session_logging	If enabled, logs of user sessions will be saved in <user_dir>/<user>/logs. Default: False
--syslog_session_logging	If enabled, logs of user sessions will be written to syslog. (default False)

Commands:

Usage: /usr/local/bin/gateone <command> [OPTIONS]

GateOne supports many different CLI 'commands' which can be used to invoke special functionality provided by plugins and their own options and most will have a --help function of their own.

Commands provided by 'gateone':

broadcast	Broadcast messages to users connected to Gate One.
install_license	Install a commercial license for Gate One.
validate_authobj	Test API authentication by validating a JSON auth object.

Commands provided by 'gateone.applications.terminal':

termlog	View terminal session logs.
---------	-----------------------------

Example command usage:

```
/usr/local/bin/gateone termlog --help
```

File Paths Gate One stores its files, temporary session information, and persistent user data in the following locations (Note: Many of these are configurable):

File/Directory	Description
authpam.py	Contains the PAM authentication Mixin used by auth.py.
auth.py	Authentication classes.
babel_gateone.cfg	Pybabel configuration for generating localized translations of Gate One's strings.
certificate.pem	The default certificate Gate One will use in SSL communications.
docs/	Gate One's documentation.
gateone.py	Gate One's primary executable/script. Also, the file containing this documentation.
gopam.py	PAM (Authentication) Python module (used by authpam.py).
i18n/	Gate One's internationalization (i18n) support and locale/translation files.
keyfile.pem	The default private key used with SSL communications.
logviewer.py	A utility to view Gate One session logs.
plugins/	(Global) Plugins go here in the form of ./plugins/<plugin name>/<plugin files directories>
settings/	All Gate One settings files live here (.conf files).
sso.py	A Kerberos Single Sign-on module for Tornado (used by auth.py)
static/	Non-dynamic files that get served to clients (e.g. gateone.js, gateone.css, etc).
templates/	Tornado template files such as index.html.
tests/	Various scripts and tools to test Gate One's functionality.
utils.py	Various supporting functions.
users/	Persistent user data in the form of ./users/<username>/<user-specific files>
users/<user>/logs	This is where session logs get stored if session_logging is set.
/tmp/gateone	Temporary session data in the form of /tmp/gateone/<session ID>/<files>
/tmp/gateone_cache	Used to store cached files such as minified JavaScript and CSS.

Running Executing Gate One is as simple as:

```
root@host:~ $ gateone
```

Note: By default Gate One will run on port 443 which requires root on most systems. Use --port=(something higher than 1024) for non-root users.

Applications and Plugins Gate One supports both *applications* and *plugins*. The difference is mostly architectural. Applications go in the gateone/applications directory while (global) plugins reside in gateone/plugins. The scope of application code applies only to the application whereas global Gate One plugin code will apply to Gate One itself and all applications.

Note: Applications may have plugins of their own (e.g. terminal/plugins).

Gate One applications and plugins can be written using any combination of the following:

- Python
- JavaScript
- CSS

Applications and Python plugins can integrate with Gate One in a number ways:

- Adding or overriding request handlers to provide custom URLs (with a given regex).
- Adding or overriding methods in `GOApplication` to handle asynchronous WebSocket "actions".
- Adding or overriding events via the `on()`, `off()`, `once()`, and `trigger()` functions.

- Delivering custom content to clients (e.g. JavaScript and CSS templates).

JavaScript and CSS plugins will be delivered over the WebSocket and cached at the client by way of a novel synchronization mechanism. Once JavaScript and CSS assets have been delivered they will only ever have to be re-delivered to clients if they have been modified (on the server). This mechanism is extremely bandwidth efficient and should allow clients to load Gate One content much more quickly than traditional HTTP GET requests.

Tip: If you install the `cssmin` and/or `slimit` Python modules all JavaScript and CSS assets will be automatically minified before being delivered to clients.

Class Docstrings

`gateone.core.server._(string)`

Wraps `server_locale.translate` so we don't get errors if loading a locale fails (or we output a message before it is initialized).

`gateone.core.server.cleanup_user_logs()`

Cleans up all user logs (everything in the user's 'logs' directory and subdirectories that ends in 'log') older than the `user_logs_max_age` setting. The log directory is assumed to be:

`user_dir/<user>/logs`

...where `user_dir` is whatever Gate One happens to have configured for that particular setting.

`gateone.core.server.cleanup_old_sessions()`

Cleans up old session directories inside the `session_dir`. Any directories found that are older than the `auth_timeout` (global gateone setting) will be removed. The modification time is what will be checked.

`gateone.core.server.clean_up()`

Regularly called via the `CLEANER` `PeriodicCallback`, calls `cleanup_user_logs` and `cleanup_old_sessions`.

Note: How often this function gets called can be controlled by adding a `cleanup_interval` setting to `10server.conf` ('gateone' section).

`gateone.core.server.policy_send_user_message(cls, policy)`

Called by `gateone_policies()`, returns True if the user is authorized to send messages to other users and if applicable, all users (broadcasts).

`gateone.core.server.policy_broadcast(cls, policy)`

Called by `gateone_policies()`, returns True if the user is authorized to broadcast messages using the `ApplicationWebSocket.broadcast()` method. It makes this determination by checking the `['gateone']['send_broadcasts']` policy.

`gateone.core.server.policy_list_users(cls, policy)`

Called by `gateone_policies()`, returns True if the user is authorized to retrieve a list of the users currently connected to the Gate One server via the `ApplicationWebSocket.list_server_users()` method. It makes this determination by checking the `['gateone']['list_users']` policy.

`gateone.core.server.gateone_policies(cls)`

This function gets registered under 'gateone' in the `ApplicationWebSocket.security` dict and is called by the `require()` decorator by way of the `policies` sub-function. It returns True or False depending on what is defined in the settings dir and what function is being called.

This function will keep track of and place limits on the following:

- Who can send messages to other users (including broadcasts).
- Who can retrieve a list of connected users.

`gateone.core.server.kill_all_sessions(timeout=False)`
Calls all 'kill_session_callbacks' attached to all SESSIONS.

If *timeout* is True, emulate a session timeout event in order to *really* kill any user sessions (to ensure things like dtach processes get killed too).

`gateone.core.server.timeout_sessions()`
Loops over the SESSIONS dict killing any sessions that haven't been used for the length of time specified in *TIMEOUT* (global). The value of *TIMEOUT* can be set in *10server.conf* or specified on the command line via the *session_timeout* value.

Applications and plugins can register functions to be called when a session times out by attaching them to the user's session inside the SESSIONS dict under 'timeout_callbacks'. The best place to do this is inside of the application's *authenticate()* function or by attaching them to the *go:authenticate* event. Examples:

```
# Imagine this is inside an application's authenticate() method:
sess = SESSIONS[self.ws.session]
# Pretend timeout_session() is a function we wrote to kill stuff
if timeout_session not in sess["timeout_session"]:
    sess["timeout_session"].append(timeout_session)
```

Note: This function is meant to be called via Tornado's *PeriodicCallback()*.

`gateone.core.server.broadcast_message(args=['/usr/local/bin/sphinx-build', '-b', 'html', '-v', '-T', '-d', 'build/doctrees', 'source', 'build/html'], message='')`
Broadcasts a given *message* to all users in Gate One. If no message is given *sys.argv* will be parsed and everything after the word 'broadcast' will be broadcast.

class `gateone.core.server.StaticHandler(application, request, **kwargs)`
An override of *tornado.web.StaticFileHandler* to ensure that the *Access-Control-Allow-Origin* header gets set correctly. This is necessary in order to support embedding Gate One into other web-based applications.

Note: Gate One performs its own origin checking so header-based access controls at the client are unnecessary.

initialize(*path*, *default_filename=None*, *use_pkg=None*)
Called automatically by the Tornado framework when the *StaticHandler* class is instantiated; handles the usual arguments with the addition of *use_pkg* which indicates that the static file should attempt to be retrieved from that package via the *pkg_resources* module instead of directly via the filesystem.

set_extra_headers(*path*)
Adds the *Access-Control-Allow-Origin* header to allow cross-origin access to static content for applications embedding Gate One. Specifically, this is necessary in order to support loading fonts from different origins.

Also sets the 'X-UA-Compatible' header to 'IE=edge' to enforce IE 10+ into standards mode when content is loaded from intranet sites.

options(*path=None*)
Replies to *OPTIONS* requests with the usual stuff (200 status, Allow header, etc). Since this is just the static file handler we don't include any extra information.

validate_absolute_path(*root, absolute_path*)

An override of `tornado.web.StaticFileHandler.validate_absolute_path()`;

Validate and returns the given *absolute_path* using `pkg_resources` if `self.use_pkg` is set otherwise performs a normal filesystem validation.

class `gateone.core.server.BaseHandler`(*application, request, **kwargs*)

A base handler that all Gate One RequestHandlers will inherit methods from.

Provides the `get_current_user()` method, sets default headers, and provides a default `options()` method that can be used for monitoring purposes and also for enumerating useful information about this Gate One server (see below for more info).

set_default_headers()

An override of `tornado.web.RequestHandler.set_default_headers()` (which is how Tornado wants you to set default headers) that adds/overrides the following headers:

Server 'GateOne'

X-UA-Compatible 'IE=edge' (forces IE 10+ into Standards mode)

get_current_user()

Tornado standard method—implemented our way.

options(*path=None*)

Replies to OPTIONS requests with the usual stuff (200 status, Allow header, etc) but also includes some useful information in the response body that lists which authentication API features we support in addition to which applications are installed. The response body is conveniently JSON-encoded:

```
user@modern-host:~ $ curl -k -X OPTIONS https://gateone.company.com/ | python -mjson.tool
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 158 100 158 0 0 6793 0 --:--:-- --:--:-- --:--:-- 7181
{
  "applications": [
    "File Transfer",
    "Terminal",
    "X11"
  ],
  "auth_api": {
    "hmacs": [
      "HMAC-SHA1",
      "HMAC-SHA256",
      "HMAC-SHA384",
      "HMAC-SHA512"
    ],
    "versions": [
      "1.0"
    ]
  }
}
```

Note: The 'Server' header does not supply the version information. This is intentional as it amounts to an unnecessary information disclosure. We don't need to make an attacker's job any easier.

class `gateone.core.server.HTTPSRedirectHandler`(*application, request, **kwargs*)

A handler to redirect clients from HTTP to HTTPS. Only used if `https_redirect` is True in Gate One's settings.

`get()`

Just redirects the client from HTTP to HTTPS

class `gateone.core.server.DownloadHandler(application, request, **kwargs)`

A `tornado.web.RequestHandler` to serve up files that wind up in a given user's `session_dir` in the 'downloads' directory. Generally speaking these files are generated by the terminal emulator (e.g. `cat somefile.pdf`) but it can be used by applications and plugins as a way to serve up all sorts of (temporary/transient) files to users.

class `gateone.core.server.MainHandler(application, request, **kwargs)`

Renders `index.html` which loads Gate One.

Will include the minified version of `gateone.js` if available as `gateone.min.js`.

class `gateone.core.server.GOApplication(ws)`

The base from which all Gate One Applications will inherit. Applications are expected to be written like so:

```
class SomeApplication(GOApplication):
    def initialize(self):
        "Called when the Application is instantiated."
        initialize_stuff()
        # Here's some good things to do in an initialize() function...
        # Register a policy-checking function:
        self.ws.security.update({'some_app': policy_checking_func})
        # Register some WebSocket actions (note the app:action naming convention)
        self.ws.actions.update({
            'some_app:do_stuff': self.do_stuff,
            'some_app:do_other_stuff': self.do_other_stuff
        })
    def open(self):
        "Called when the connection is established."
        # Setup whatever is necessary for session tracking and whatnot.
    def authenticate(self):
        "Called when the user *successfully* authenticates."
        # Here's the best place to instantiate things, send the user
        # JavaScript/CSS files, and similar post-authentication details.
    def on_close(self):
        "Called when the connection is closed."
        # This is a good place to halt any background/periodic operations.
```

GOApplications will be automatically imported into Gate One and registered appropriately as long as they follow the following conventions:

- The application and its module(s) should live inside its own directory inside the 'applications' directory. For example, `/opt/gateone/applications/some_app/some_app.py`
- Subclasses of `GOApplication` must be added to an `apps` global (list) inside of the application's module(s) like so: `apps = [SomeApplication]` (usually a good idea to put that at the very bottom of the module).

Note: All `.py` modules inside of the application's main directory will be imported even if they do not contain or register a `GOApplication`.

Tip: You can add command line arguments to Gate One by calling `tornado.options.define()` anywhere in your application's global namespace. This works because the `define()` function registers options in Gate One's global namespace (as `tornado.options.options`) and Gate One imports application modules before it evaluates command line arguments.

initialize()

Called by `ApplicationWebSocket.open()` after `__init__()`. `GOApplications` can override this function to perform their own actions when the Application is initialized (happens just before the WebSocket is opened).

open()

Called by `ApplicationWebSocket.open()` after the WebSocket is opened. `GOApplications` can override this function to perform their own actions when the WebSocket is opened.

on_close()

Called by `ApplicationWebSocket.on_close()` after the WebSocket is closed. `GOApplications` can override this function to perform their own actions when the WebSocket is closed.

add_handler(*pattern, handler, **kwargs*)

Adds the given *handler* (`tornado.web.RequestHandler`) to the Tornado Application (`self.ws.application`) to handle URLs matching *pattern*. If given, *kwargs* will be added to the `tornado.web.URLSpec` when the complete handler is assembled.

Note: If the *pattern* does not start with the configured `url_prefix` it will be automatically prepended.

add_timeout(*timeout, func*)

A convenience function that calls the given *func* after *timeout* using `self.io_loop.add_timeout()` (which uses `tornado.ioloop.IOLoop.add_timeout()`).

The given *timeout* may be a `datetime.timedelta` or a string compatible with `utils.convert_to_timedelta` such as, "5s" or "10m".

class `gateone.core.server.ApplicationWebSocket(application, request, **kwargs)`

The main WebSocket interface for Gate One, this class is setup to call WebSocket 'actions' which are methods registered in `self.actions`. Methods that are registered this way will be exposed and directly callable over the WebSocket.

classmethod `file_checker()`

A `Tornado.IOLoop.PeriodicCallback` that regularly checks all files registered in the `ApplicationWebSocket.watched_files` dict for changes.

If changes are detected the corresponding function(s) in `ApplicationWebSocket.file_update_funcs` will be called.

classmethod `watch_file(path, func)`

A classmethod that registers the given file *path* and *func* in `ApplicationWebSocket.watched_files` and `ApplicationWebSocket.file_update_funcs`, respectively. The given *func* will be called (by `ApplicationWebSocket.file_checker`) whenever the file at *path* is modified.

classmethod `load_prefs()`

Loads all of Gate One's settings from `options.settings_dir` into `cls.prefs`.

Note: This classmethod gets called automatically whenever a change is detected inside Gate One's `settings_dir`.

classmethod `broadcast_file_update()`

Called when there's an update to the `broadcast_file` (e.g. `<session_dir>/broadcast`); broadcasts its contents to all connected users. The message will be displayed via the `GateOne.Visual.displayMessage()` function at the client and can be formatted with HTML. For this reason it is important to strictly control write access to the broadcast file.

Note: Under normal circumstances only root (or the owner of the gateone.py process) can enter and/or write to files inside Gate One's `session_dir` directory.

The path to the broadcast file can be configured via the `broadcast_file` setting which can be placed anywhere under the 'gateone' application/scope (e.g. inside `10server.conf`). The setting isn't there by default but you can simply add it if you wish:

```
"broadcast_file": "/whatever/path/you/want/broadcast"
```

Tip: Want to broadcast a message to all the users currently connected to Gate One? Just `sudo echo "your message" > /tmp/gateone/broadcast`.

initialize(*apps=None, **kwargs*)

This gets called by the Tornado framework when `ApplicationWebSocket` is instantiated. It will be passed the list of *apps* (Gate One applications) that are assigned inside the `GOApplication` object. These `GOApplication`'s will be instantiated and stored in `self.apps`.

These *apps* will be mutated in-place so that `self` will refer to the current instance of `ApplicationWebSocket`. Kind of like a dynamic mixin.

send_extra()

Sends any extra JS/CSS files placed in Gate One's 'static/extra' directory. Can be useful if you want to use Gate One's file synchronization and caching capabilities in your app.

Note: You may have to create the 'static/extra' directory before putting files in there.

allow_draft76()

By overriding this function we're allowing the older version of the WebSockets protocol. As long as communications happens over SSL there shouldn't be any security concerns with this. This is mostly to support iOS Safari.

get_current_user()

Mostly identical to the function of the same name in `MainHandler`. The difference being that when API authentication is enabled the `WebSocket` will expect and perform its own auth of the client.

write_binary(*message*)

Writes the given *message* to the `WebSocket` in binary mode (opcode 0x02). Binary `WebSocket` messages are handled differently from regular messages at the client (they use a completely different 'action' mechanism). For more information see the JavaScript developer documentation.

check_origin(*origin*)

Checks if the given *origin* matches what's been set in Gate One's "origins" setting (usually in `10server.conf`). The *origin* will first be checked for an exact match in the "origins" setting but if that fails each valid origin will be evaluated as a regular expression (if it's not a valid hostname) and the given *origin* will be checked against that.

Returns True if *origin* is valid.

Note: If '*' is in the "origins" setting (anywhere) all origins will be allowed.

open()

Called when a new `WebSocket` is opened. Will deny access to any origin that is not defined in `self.settings['origin']`. Also sends any relevant localization data (JavaScript) to the client and calls the `open()` method of any and all enabled Applications.

This method kicks off the process that sends keepalive pings/checks to the client (A Periodic-Callback set as `self.pinger`).

This method also sets the following instance attributes:

- `self.client_id`: Unique identifier for this instance.
- `self.base_url`: The base URL (e.g. <https://foo.com/gateone/>) used to access Gate One.

Triggers the `go:open` event.

Note: `self.settings` comes from the Tornado framework and includes most command line arguments and the settings from the `settings_dir` that fall under the “gateone” scope. It is not the same thing as `self.prefs` which includes *all* of Gate One’s settings (including settings for other applications and scopes).

`on_message(message)`

Called when we receive a message from the client. Performs some basic validation of the message, decodes it (JSON), and finally calls an appropriate WebSocket action (registered method) with the message contents.

`on_close()`

Called when the client terminates the connection. Also calls the `on_close()` method of any and all enabled Applications.

Triggers the `go:close` event.

`on_pong(timestamp)`

Records the latency of clients (from the server’s perspective) via a log message.

Note: This is the pong specified in the WebSocket protocol itself. The pong method is a Gate One-specific implementation.

`pong(timestamp)`

Attached to the `go:ping` WebSocket action; responds to a client’s ping by returning the value (*timestamp*) that was sent. This allows the client to measure the round-trip time of the WebSocket.

Note: This is a WebSocket action specific to Gate One. It

`api_auth(auth_obj)`

If the *auth_obj* dict validates, returns the user dict and sets `self.current_user`. If it doesn’t validate, returns `False`.

This function also takes care of creating the user’s directory if it does not exist and creating/updating the user’s ‘session’ file (which just stores metadata related to their session).

Example usage:

```
auth_obj = {
    'api_key': 'MjkwYzc3MDI2MjhhNGZkNDg1MjJkODgyYjBmN2MyMTM4M',
    'upn': 'joe@company.com',
    'timestamp': '1323391717238',
    'signature': '<gibberish>',
    'signature_method': 'HMAC-SHA1',
    'api_version': '1.0'
}
result = self.api_auth(auth_obj)
```

See also:

API Authentication documentation.

Here's a rundown of the required *auth_obj* parameters:

api_key The first half of what gets generated when you run `gateone --new_api_key` (the other half is the secret).

upn The `userPrincipalName` (aka username) of the user being authenticated.

timestamp A 13-digit "time since the epoch" JavaScript-style timestamp. Both integers and strings are accepted. Example JavaScript: `var timestamp = new Date().getTime()`

signature The HMAC signature of the combined *api_key*, *upn*, and *timestamp*; hashed using the secret associated with the given *api_key*.

signature_method The hashing algorithm used to create the *signature*. Currently this must be one of "HMAC-SHA1", "HMAC-SHA256", "HMAC-SHA384", or "HMAC-SHA512"

api_version Which version of the authentication API to use when performing authentication. Currently the only supported version is '1.0'.

Note: Any additional key/value pairs that are included in the *auth_obj* will be assigned to the `self.current_user` object. So if you're embedding Gate One and wish to associate extra metadata with the user you may do so via the API authentication process.

authenticate(settings)

Authenticates the client by first trying to use the 'gateone_user' cookie or if Gate One is configured to use API authentication it will use *settings['auth']*. Additionally, it will accept *settings['container']* and *settings['prefix']* to apply those to the equivalent properties (`self.container` and `self.prefix`).

If *settings['url']* is provided it will be used to update `self.base_url` (so that we can correct for situations where Gate One is running behind a reverse proxy with a different protocol/URL than what the user used to connect).

Note: 'container' refers to the element on which Gate One was initialized at the client (e.g. `#gateone`). 'prefix' refers to the string that will be prepended to all Gate One element IDs when added to the web page (to avoid namespace conflicts). Both these values are only used when generating CSS templates.

If *settings['location']* is something other than 'default' all new application instances will be associated with the given (string) value. These applications will be treated separately so they can exist in a different browser tab/window.

Triggers the `go:authenticate` event.

_start_session_watcher(restart=False)

Starts up the `SESSION_WATCHER` (assigned to that global) `PeriodicCallback` that regularly checks for user sessions that have timed out via the `timeout_sessions()` function and cleans them up (shuts down associated processes).

The interval in which it performs this check is controlled via the `session_timeout_check_interval` setting. This setting is not included in Gate One's `10server.conf` by default but can be added if needed to override the default value of 30 seconds. Example:

```
{
  "*": {
    "gateone": {
      "session_timeout_check_interval": "30s"
    }
  }
}
```

`_start_cleaner()`

Starts up the CLEANER (assigned to that global) `PeriodicCallback` that regularly checks for and deletes expired user logs (e.g. terminal session logs or anything in the `<user_dir>/<user>/logs` dir) and old session directories via the `cleanup_user_logs()` and `cleanup_old_sessions()` functions.

The interval in which it performs this check is controlled via the `cleanup_interval` setting. This setting is not included in Gate One's `10server.conf` by default but can be added if needed to override the default value of 5 minutes. Example:

```
{
  "*": {
    "gateone": {
      "cleanup_interval": "5m"
    }
  }
}
```

`_start_file_watcher()`

Starts up the `ApplicationWebSocket.file_watcher` `PeriodicCallback` (which regularly calls `ApplicationWebSocket.file_checker()` and immediately starts it watching the broadcast file for changes (if not already watching it).

The path to the broadcast file defaults to `'settings_dir/broadcast'` but can be changed via the `'broadcast_file'` setting. This setting is not included in Gate One's `10server.conf` by default but can be added if needed to override the default value. Example:

```
{
  "*": {
    "gateone": {
      "broadcast_file": "/some/path/to/broadcast"
    }
  }
}
```

Tip: You can send messages to all users currently connected to the Gate One server by writing text to the broadcast file. Example: `sudo echo "Server will be rebooted as part of regularly scheduled maintenance in 5 minutes. Please save your work." > /tmp/gateone/broadcast`

The interval in which it performs this check is controlled via the `file_check_interval` setting. This setting is not included in Gate One's `10server.conf` by default but can be added if needed to override the default value of 5 seconds. Example:

```
{
  "*": {
    "gateone": {
      "file_check_interval": "5s"
    }
  }
}
```

```
}
}
```

list_applications()

Sends a message to the client indicating which applications and sub-applications are available to the user.

Note: What's the difference between an "application" and a "sub-application"? An "application" is a `GOApplication` like `app_terminal.TerminalApplication` while a "sub-application" would be something like "SSH" or "nethack" which runs inside the parent application.

render_style(style_path, force=False, **kwargs)

Renders the CSS template at *style_path* using *kwargs* and returns the path to the rendered result. If the given style has already been rendered the existing cache path will be returned.

If *force* is `True` the stylesheet will be rendered even if it already exists (cached).

This method also cleans up older versions of the same rendered template.

get_theme(settings)

Sends the theme stylesheets matching the properties specified in *settings* to the client. *settings* must contain the following:

- container** - The element Gate One resides in (e.g. 'gateone')
- prefix** - The string being used to prefix all elements (e.g. 'go_')
- theme** - The name of the CSS theme to be retrieved.

Note: This will send the theme files for all applications and plugins that have a matching stylesheet in their 'templates' directory.

cache_cleanup(message)

Attached to the `go:cache_cleanup` WebSocket action; rifles through the given list of *message['filenames']* from the client and sends a `go:cache_expired` WebSocket action to the client with a list of files that no longer exist in `self.file_cache` (so it can clean them up).

file_request(files_or_hash, use_client_cache=True)

Attached to the `go:file_request` WebSocket action; minifies, caches, and finally sends the requested file to the client. If *use_client_cache* is `False` the client will be instructed not to cache the file. Example message from the client requesting a file:

```
GateOne.ws.send(JSON.stringify({
  'go:file_request': {'some_file.js'}}));
```

Note: In reality 'some_file.js' will be a unique/unguessable hash.

Optionally, *files_or_hash* may be given as a list or tuple and all the requested files will be sent. Files will be cached after being minified until a file is modified or Gate One is restarted.

If the `slimit` module is installed JavaScript files will be minified before being sent to the client.

If the `cssmin` module is installed CSS files will be minified before being sent to the client.

send_file(filepath, kind='misc', **metadata)

Tells the client to perform a sync of the file at the given *filepath*. The *kind* should only be one of 'html' or 'misc' for HTML templates and everything else, respectively.

Any additional keyword arguments provided via *metadata* will be stored in the client-side file-Cache database.

Note: This kind of file sending *always* uses the client-side cache.

send_js_or_css(*paths_or_fileobj*, *kind*, *element_id=None*, *requires=None*, *media='screen'*, *filename=None*, *force=False*)

Initiates a file synchronization of the given *paths_or_fileobj* with the client to ensure it has the latest version of the file(s).

The *kind* argument must be one of 'js' or 'css' to indicate JavaScript or CSS, respectively.

Optionally, *element_id* may be provided which will be assigned to the <script> or <style> tag that winds up being created (only works with single files).

Optionally, a *requires* string or list/tuple may be given which will ensure that the given file gets loaded after any dependencies.

Optionally, a *media* string may be provided to specify the 'media=' value when creating a <style> tag to hold the given CSS.

Optionally, a *filename* string may be provided which will be used instead of the name of the actual file when file synchronization occurs. This is useful for multi-stage processes (e.g. rendering templates) where you wish to preserve the original filename. Just be aware that if you do this the given *filename* must be unique.

If *force* is True the file will be synchronized regardless of the 'send_js' or 'send_css' settings in your global Gate One settings.

send_js(*path*, ***kwargs*)

A shortcut for `self.send_js_or_css(path, 'js', **kwargs)`.

send_css(*path*, ***kwargs*)

A shortcut for `self.send_js_or_css(path, 'css', **kwargs)`

wrap_and_send_js(*js_path*, *exports={}*, ***kwargs*)

Wraps the JavaScript code at *js_path* in a (JavaScript) sandbox which exports whatever global variables are provided via *exports* then minifies, caches, and sends the result to the client.

The provided *kwargs* will be passed to the `ApplicationWebSocket.send_js` method.

The *exports* dict needs to be in the following format:

```
exports = {  
    "global": "export_name"  
}
```

For example, if you wanted to use underscore.js but didn't want to overwrite the global `_` variable (if already being used by a parent web application):

```
exports = {"_": "GateOne._"}  
self.wrap_and_send_js('/path/to/underscore.js', exports)
```

This would result in the `"_"` global being exported as `"GateOne._"`. In other words, this is what will end up at the bottom of the wrapped JavaScript just before the end of the sandbox:

```
window.GateOne._ = _;
```

This method should make it easy to include any given JavaScript library without having to worry (as much) about namespace conflicts.

Note: You don't have to prefix your export with 'GateOne'. You can export the global with whatever name you like.

render_and_send_css(*css_path*, *element_id=None*, *media='screen'*, ***kwargs*)

Renders, caches (in the *cache_dir*), and sends a stylesheet template at the given *css_path*. The template will be rendered with the following keyword arguments:

```
container = self.container
prefix = self.prefix
url_prefix = self.settings['url_prefix']
**kwargs
```

Returns the path to the rendered template.

Note: If you want to serve Gate One's CSS via a different mechanism (e.g. nginx) this functionality can be completely disabled by adding "send_css": false to gateone/settings/10server.conf

send_plugin_static_files(*entry_point*, *requires=None*)

Sends all plugin .js and .css files to the client that exist inside the /static/ directory for given *entry_point*. The policies that apply to the current user will be used to determine whether and which static files will be sent.

If *requires* is given it will be passed along to *self.send_js()*.

Note: If you want to serve Gate One's JavaScript via a different mechanism (e.g. nginx) this functionality can be completely disabled by adding "send_js": false to gateone/settings/10server.conf

enumerate_themes()

Returns a JSON-encoded object containing the installed themes and text color schemes.

set_locales(*locales*)

Attached to the 'go:set_locales' WebSocket action; sets *self.user_locales* to the given *locales* and calls *ApplicationWebSocket.send_js_translation* to deliver the best-match JSON-encoded translation to the client (if available).

The *locales* argument may be a string or a list. If a string, *self.user_locales* will be set to a list with the given locale as the first (and only) item. If *locales* is a list *self.user_locales* will simply be replaced.

send_js_translation(*package='gateone'*, *path=None*, *locales=None*)

Sends a message to the client containing a JSON-encoded table of strings that have been translated to the user's locale. The translation file will be retrieved via the *pkg_resources.resource_string* function using the given *package* with a default path (if not given) of, *'/i18n/{locale}/LC_MESSAGES/gateone_js.json'*. For example:

```
self.send_js_translation(package="gateone.plugins.myplugin")
```

Would result in the *pkg_resources.resource_string* function being called like so:

```
path = '/i18n/{locale}/LC_MESSAGES/gateone_js.json'.format(
    locale=locale)
translation = pkg_resources.resource_string(
    "gateone.plugins.myplugin", path)
```

If providing a custom *path* be sure it includes the '{locale}' portion so the correct translation will be chosen. As an example, your plugin might store locales inside a 'translations' directory

with each locale JSON translation file named, '*<locale>/myplugin.json*'. The correct *path* for that would be: '*/translations/{locale}/myplugin.json*'

If no *locales* (may be list or string) is given the *self.user_locales* variable will be used.

This method will attempt to find the closest locale if no direct match can be found. For example, if *locales*=="fr" the 'fr_FR' locale would be used. If *locales* is a list, the first matching locale will be used.

Note: Translation files must be the result of a *pojson /path/to/translation.po* conversion.

send_message(*message*, *session*=None, *upn*=None)

Sends the given *message* to the client using the *go:user_message* WebSocket action at the currently-connected client.

If *upn* is provided the *message* will be sent to all users with a matching 'upn' value.

If *session* is provided the message will be sent to all users with a matching session ID. This is useful in situations where all users share the same 'upn' (i.e. ANONYMOUS).

if *upn* is 'AUTHENTICATED' all users will get the message.

classmethod _deliver(*message*, *upn*='AUTHENTICATED', *session*=None)

Writes the given *message* (string) to all users matching *upn* using the *write_message()* function. If *upn* is not provided or is "AUTHENTICATED", will send the *message* to all users.

Alternatively a *session* ID may be specified instead of a *upn*. This is useful when more than one user shares a UPN (i.e. ANONYMOUS).

classmethod _list_connected_users()

Returns a tuple of user objects representing the users that are currently connected (and authenticated) to this Gate One server.

license_info()

Returns the contents of the *__license_info__* dict to the client as a JSON-encoded message.

class *gateone.core.server.ErrorHandler*(*application*, *request*, *status_code*)

Generates an error response with *status_code* for all requests.

gateone.core.server.validate_authobj(*args*=['/usr/local/bin/sphinx-build', '-b', 'html', '-v', '-T', '-d', 'build/doctrees', 'source', 'build/html'])

Handles the 'validate_authobj' CLI command. Takes a JSON object as the only argument (must be inside single quotes) and validates the singature using the same mechanism as *ApplicationWebSocket.api_auth*. Example usage:

```
user@modern-host:~ $ gateone validate_authobj '{"upn": "jdoe@company.com", "signature_method": "HMAC-SHA1", "timestamp": 1234567890}'
API Authentication Successful!
```

gateone.core.server.install_license(*args*=['/usr/local/bin/sphinx-build', '-b', 'html', '-v', '-T', '-d', 'build/doctrees', 'source', 'build/html'])

Handles the 'install_license' CLI command. Just installs the license at the path given via *sys.argv[1]* (first argument after the 'install_license' command).

gateone.core.server.validate_licenses(*licenses*)

Given a *licenses* dict, logs and removes any licenses that have expired or have invalid signatures. It then sets the *__license_info__* global with the updated dict. Example license dict:

```
{
  "11252c41-d3cd-45b7-929b-4a3baedcc152": {
    "product": "gateone",
    "customer": "ACME, Inc",
```



```

        "users": 250,
        "expires": 1441071222,
        "license_format": "1.0",
        "signature": "<gobbledygook>",
        "signature_method": "secp256k1"
    }
}

```

Note: Signatures will be validated against Liftoff Software’s public key.

terminal.py - A Pure Python Terminal Emulator

termio.py - Terminal Input/Output Module

utils.py - Supporting Functions

Gate One Utility Functions and Classes

exception gateone.core.utils.MimeTypeFail

Raised by create_data_uri if the mimetype of a file could not be guessed.

exception gateone.core.utils.SSLGenerationError

Raised by gen_self_signed_ssl if an error is encountered generating a self-signed SSL certificate.

exception gateone.core.utils.ChownError

Raised by recursive_chown if an OSError is encountered while trying to recursively chown a directory.

class gateone.core.utils.AutoExpireDict(*args, **kwargs)

An override of Python’s dict that expires keys after a given *_expire_timeout* timeout (datetime.timedelta). The default expiration is one hour. It is used like so:

```

>>> expiring_dict = AutoExpireDict(timeout=datetime.timedelta(minutes=10))
>>> expiring_dict['somekey'] = 'some value'
>>> # You can see when this key was created:
>>> print(expiring_dict.creation_times['somekey'])
2013-04-15 18:44:18.224072

```

10 minutes later your key will be gone:

```

>>> 'somekey' in expiring_dict
False

```

The ‘timeout’ may be given as a datetime.timedelta object or a string like, “1d”, “30s” (will be passed through the convert_to_timedelta function).

By default AutoExpireDict will check for expired keys every 30 seconds but this can be changed by setting the ‘interval’:

```

>>> expiring_dict = AutoExpireDict(interval=5000) # 5 secs
>>> # Or to change it after you've created one:
>>> expiring_dict.interval = "10s"

```

The ‘interval’ may be an integer, a datetime.timedelta object, or a string such as ‘10s’ or ‘5m’ (will be passed through the convert_to_timedelta function).

If there are no keys remaining the tornado.ioloop.PeriodicCallback (self._key_watcher) that checks expiration will be automatically stopped. As soon as a new key is added it will be started

back up again.

Note: Only works if there's a running instances of `tornado.ioloop.IOLoop`.

timeout

A property that controls how long a key will last before being automatically removed. May be given as a `datetime.timedelta` object or a string like, "1d", "30s" (will be passed through the `convert_to_timedelta` function).

interval

A property that controls how often we check for expired keys. May be given as milliseconds (integer), a `datetime.timedelta` object, or a string like, "1d", "30s" (will be passed through the `convert_to_timedelta` function).

renew(*key*)

Resets the timeout on the given *key*; like it was just created.

update(args*, ***kwargs*)**

An override that calls `self.renew()` for every key that gets updated.

clear()

An override that empties `self.creation_times` and calls `self._key_watcher.stop()`.

_timeout_checker()

Walks `self` and removes keys that have passed the expiration point.

class `gateone.core.utils.memoize(fn, timeout=None)`

A memoization decorator that works with multiple arguments as well as unhashable arguments (e.g. dicts). It also self-expires any memoized calls after the `timedelta` specified via *timeout*.

If a *timeout* is not given memoized information will be discarded after five minutes.

Note: Expiration checks will be performed every 30 seconds.

`gateone.core.utils.noop(*args, **kwargs)`

Do nothing (i.e. "No Operation")

`gateone.core.utils.debug_info(name, *args, **kwargs)`

This function returns a string like this:

```
>>> debug_info('some_function', 5, 10, foo="bar")
'some_function(5, 10, foo="bar")'
```

Primarily aimed at debugging.

`gateone.core.utils.write_pid(path)`

Writes our PID to *path*.

`gateone.core.utils.read_pid(path)`

Reads our current PID from *path*.

`gateone.core.utils.remove_pid(path)`

Removes the PID file at *path*.

`gateone.core.utils.shell_command(cmd, timeout_duration=5)`

Resets the SIGCHLD signal handler (if necessary), executes *cmd* via `getstatusoutput()`, then re-enables the SIGCHLD handler (if it was set to something other than SIG_DFL). Returns the result of `getstatusoutput()` which is a tuple in the form of:

`(exitstatus, output)`

If the command takes longer than *timeout_duration* seconds, it will be auto-killed and the following will be returned:

```
(255, _("ERROR: Timeout running shell command"))
```

gateone.core.utils.**json_encode**(*obj*)

On some platforms (CentOS 6.2, specifically) `tornado.escape.json_decode` doesn't seem to work just right when it comes to returning unicode strings. This is just a wrapper that ensures that the returned string is unicode.

gateone.core.utils.**gen_self_signed_ssl**(*path=None*)

Generates a self-signed SSL certificate using `pyOpenSSL` or the `openssl` command depending on what's available. The resulting key/certificate will use the RSA algorithm at 4096 bits.

gateone.core.utils.**gen_self_signed_openssl**(*path=None*)

This method will generate a secure self-signed SSL key/certificate pair (using the `openssl` command) saving the result as 'certificate.pem' and 'keyfile.pem' to *path*. If *path* is not given the result will be saved in the current working directory. The certificate will be valid for 10 years.

Note: The self-signed certificate will utilize 4096-bit RSA encryption.

gateone.core.utils.**gen_self_signed_pyopenssl**(*notAfter=None, path=None*)

This method will generate a secure self-signed SSL key/certificate pair (using `pyOpenSSL`) saving the result as 'certificate.pem' and 'keyfile.pem' in *path*. If *path* is not given the result will be saved in the current working directory. By default the certificate will be valid for 10 years but this can be overridden by passing a valid timestamp via the *notAfter* argument.

Examples:

```
>>> gen_self_signed_ssl(60 * 60 * 24 * 365) # 1-year certificate
>>> gen_self_signed_ssl() # 10-year certificate
```

Note: The self-signed certificate will utilize 4096-bit RSA encryption.

gateone.core.utils.**none_fix**(*val*)

If *val* is a string that ultimately means 'none', return `None`. Otherwise return *val* as-is. Examples:

```
>>> none_fix('none')
None
>>> none_fix('0')
None
>>> none_fix('whatever')
'whatever'
```

gateone.core.utils.**str2bool**(*val*)

Converts strings like, 'false', 'true', '0', and '1' into their boolean equivalents (in Python). If no logical match is found, return `False`. Examples:

```
>>> str2bool('false')
False
>>> str2bool('1')
True
>>> str2bool('whatever')
False
```

gateone.core.utils.**generate_session_id**()

Returns a random, 45-character session ID. Example:

```
>>> generate_session_id()
"NzY4YzFmNDdhMTM1NDg3Y2FkZmZkMWJmYjYzNjBjM2Y5O"
>>>
```

`gateone.core.utils.mkdir_p(path)`
Pythonic version of “`mkdir -p`”. Example equivalents:

```
>>> mkdir_p('/tmp/test/testing') # Does the same thing as...
>>> from subprocess import call
>>> call('mkdir -p /tmp/test/testing')
```

Note: This doesn’t actually call any external commands.

`gateone.core.utils.cmd_var_swap(cmd, **kwargs)`
Returns `cmd` with `%variable%` replaced with the keys/values passed in via `kwargs`. This function is used by Gate One’s Terminal application to swap the following Gate One variables in defined terminal ‘commands’:

<code>%SESSION%</code>	<code>session</code>
<code>%SESSION_DIR%</code>	<code>session_dir</code>
<code>%SESSION_HASH%</code>	<code>session_hash</code>
<code>%USERDIR%</code>	<code>user_dir</code>
<code>%USER%</code>	<code>user</code>
<code>%TIME%</code>	<code>time</code>

This allows for unique or user-specific values to be swapped into command line arguments like so:

```
ssh_connect.py -M -S '%SESSION%/%SESSION%/%r@%L:%p'
```

Could become:

```
ssh_connect.py -M -S '/tmp/gateone/NWI0YzYxNzAwMTA3NGYyZmI0OWJmODczYmQyMjQwMDYwM/%r@%L:%p'
```

Here’s an example:

```
>>> cmd = "echo '%FOO% %BAR%'"
>>> cmd_var_swap(cmd, foo="FOOYEAH, ", bar="BAR NONE!")
"echo 'FOOYEAH, BAR NONE!'"
```

Note: The variables passed into this function via `kwargs` are case insensitive. `cmd_var_swap(cmd, session=var)` would produce the same output as `cmd_var_swap(cmd, SESSION=var)`.

`gateone.core.utils.short_hash(to_shorten)`
Converts `to_shorten` into a really short hash dependent on the length of `to_shorten`. The result will be safe for use as a file name.

Note: Collisions are possible but *highly* unlikely because of how this method is typically used.

`gateone.core.utils.random_words(n=1)`
Returns `n` random English words (as a tuple of strings) from the `english_wordlist.txt` file (bundled with Gate One).

Note: In Python 2 the words will be Unicode strings.

`gateone.core.utils.get_process_tree(parent_pid)`
Returns a list of child pids that were spawned from `parent_pid`.

Note: Will include `parent_pid` in the output list.

`gateone.core.utils.kill_dtached_proc_bsd(session, location, term)`

A BSD-specific implementation of `kill_dtached_proc` since Macs don't have `/proc`. Seems simpler than `kill_dtached_proc()` but actually having to call a subprocess is less efficient (due to the sophisticated signal handling required by `shell_command()`).

`gateone.core.utils.killall_bsd(session_dir, pid_file=None)`

A BSD-specific version of `killall` since Macs don't have `/proc`.

Note: `pid_file` is not used by this function. It is simply here to provide compatibility with `killall`.

`gateone.core.utils.kill_session_processes(session)`

Kills all processes that match a given *session* (which is a unique, 45-character string).

`gateone.core.utils.entry_point_files(ep_group, enabled=None)`

Given an entry point group name (*ep_group*), returns a dict of available Python, JS, and CSS plugins for Gate One:

```
{
    'css': ['editor/static/codemirror.css'],
    'js': ['editor/static/codemirror.js', 'editor/static/editor.js'],
    'py': [<module 'gateone.plugins.editor' from 'gateone/plugins/editor/__init__.pyc'>]
}
```

Optionally, the returned dict will include only those modules and files for plugins in the *enabled* list (if given).

Note: Python plugins will be imported automatically as part of the discovery process.

To do this it uses the `pkg_resources` module from `setuptools`. For plugins to be imported correctly they need to register themselves using the given entry point group (*ep_group*) in their `setup.py`. Gate One (currently) uses the following entry point group names:

- `go_plugins`
- `go_applications`
- `go_applications_plugins`

...but this function can return the JS, CSS, and Python modules for any entry point that uses the same `module_name/static/` layout.

`gateone.core.utils.load_modules(modules)`

Given a list of Python *modules*, imports them.

Note: Assumes they're all in `sys.path`.

`gateone.core.utils.merge_handlers(handlers)`

Takes a list of Tornado *handlers* like this:

```
[
    (r"/", MainHandler),
    (r"/ws", TerminalWebSocket),
    (r"/auth", AuthHandler),
    (r"/style", StyleHandler),
    ...
]
```

```
(r"/style", SomePluginHandler),  
]
```

...and returns a list with duplicate handlers removed; giving precedence to handlers with higher indexes. This allows plugins to override Gate One's default handlers. Given the above, this is what would be returned:

```
[  
    (r"/", MainHandler),  
    (r"/ws", TerminalWebSocket),  
    (r"/auth", AuthHandler),  
    ...  
    (r"/style", SomePluginHandler),  
]
```

This example would replace the default `/style` handler with `SomePluginHandler`; overriding Gate One's default `StyleHandler`.

`gateone.core.utils.convert_to_timedelta(time_val)`

Given a *time_val* (string) such as `'5d'`, returns a `datetime.timedelta` object representing the given value (e.g. `timedelta(days=5)`). Accepts the following `'<num><char>'` formats:

Character	Meaning	Example
(none)	Milliseconds	'500' -> 500 Milliseconds
s	Seconds	'60s' -> 60 Seconds
m	Minutes	'5m' -> 5 Minutes
h	Hours	'24h' -> 24 Hours
d	Days	'7d' -> 7 Days
M	Months	'2M' -> 2 Months
y	Years	'10y' -> 10 Years

Examples:

```
>>> convert_to_timedelta('7d')  
datetime.timedelta(7)  
>>> convert_to_timedelta('24h')  
datetime.timedelta(1)  
>>> convert_to_timedelta('60m')  
datetime.timedelta(0, 3600)  
>>> convert_to_timedelta('120s')  
datetime.timedelta(0, 120)
```

`gateone.core.utils.convert_to_bytes(size_val)`

Given a *size_val* (string) such as `'100M'`, returns an integer representing an equivalent amount of bytes. Accepts the following `'<num><char>'` formats:

Character	Meaning	Example
B (or none)	Bytes	'100' or '100b' -> 100
K	Kilobytes	'1k' -> 1024
M	Megabytes	'1m' -> 1048576
G	Gigabytes	'1g' -> 1073741824
T	Terabytes	'1t' -> 1099511627776
P	Petabytes	'1p' -> 1125899906842624
E	Exabytes	'1e' -> 1152921504606846976
Z	Zettabytes	'1z' -> 1180591620717411303424L
Y	Yottabytes	'1y' -> 1208925819614629174706176L

Note: If no character is given the *size_val* will be assumed to be in bytes.

Tip: All characters will be converted to upper case before conversion (case-insensitive).

Examples:

```
>>> convert_to_bytes('2M')
2097152
>>> convert_to_bytes('2g')
2147483648
```

`gateone.core.utils.total_seconds(td)`

Given a *timedelta* (*td*) return an integer representing the equivalent of Python 2.7's `datetime.timedelta.total_seconds()`.

`gateone.core.utils.process_opt_esc_sequence(chars)`

Parse the *chars* passed from `terminal.Terminal` by way of the special, optional escape sequence handler (e.g. '`<plugin>|<text>`') into a tuple of (`<plugin name>`, `<text>`). Here's an example:

```
>>> process_opt_esc_sequence('ssh|user@host:22')
('ssh', 'user@host:22')
```

`gateone.core.utils.raw(text, replacement_dict=None)`

Returns *text* as a string with special characters replaced by visible equivalents using *replacement_dict*. If *replacement_dict* is `None` or `False` the global `REPLACEMENT_DICT` will be used. Example:

```
>>> test = '\x1b]0;Some xterm title\x07'
>>> print(raw(test))
'^[]0;Some title^G'
```

`gateone.core.utils.create_data_uri(filepath, mimetype=None)`

Given a file at *filepath*, return that file as a data URI.

Raises a `MimeTypeFail` exception if the *mimetype* could not be guessed.

`gateone.core.utils.human_readable_bytes(nbytes)`

Returns *nbytes* as a human-readable string in a similar fashion to how it would be displayed by `ls -lh` or `df -h`.

`gateone.core.utils.which(binary, path=None)`

Returns the full path of *binary* (string) just like the 'which' command. Optionally, a *path* (colon-delimited string) may be given to use instead of `os.environ['PATH']`.

`gateone.core.utils.touch(path)`

Emulates the 'touch' command by creating the file at *path* if it does not exist. If the file exist its modification time will be updated.

`gateone.core.utils.timeout_func(func, args=(), kwargs={}, timeout_duration=10, default=None)`

Sets a timeout on the given function, passing it the given args, kwargs, and a *default* value to return in the event of a timeout. If *default* is a function that function will be called in the event of a timeout.

`gateone.core.utils.valid_hostname(hostname, allow_underscore=False)`

Returns True if the given *hostname* is valid according to RFC rules. Works with Internationalized Domain Names (IDN) and optionally, hostnames with an underscore (if *allow_underscore* is True).

The rules for hostnames:

- Must be less than 255 characters.
- Individual labels (separated by dots) must be <= 63 characters.

- Only the ASCII alphabet (A-Z) is allowed along with dashes (-) and dots (.).
- May not start with a dash or a dot.
- May not end with a dash.
- If an IDN, when converted to Punycode it must comply with the above.

IP addresses will be validated according to their well-known specifications.

Examples:

```
>>> valid_hostname('foo.bar.com.') # Standard FQDN
True
>>> valid_hostname('2foo') # Short hostname
True
>>> valid_hostname('-2foo') # No good: Starts with a dash
False
>>> valid_hostname('host_a') # No good: Can't have underscore
False
>>> valid_hostname('host_a', allow_underscore=True) # Now it'll validate
True
>>> valid_hostname(u'foo.jp') # Example valid IDN
True
```

`gateone.core.utils.recursive_chown(path, uid, gid)`
Emulates 'chown -R uid:gid path' in pure Python

`gateone.core.utils.check_write_permissions(user, path)`
Returns True if the given *user* has write permissions to *path*. *user* can be a UID (int) or a username (string).

`gateone.core.utils.bind(function, self)`
Will return *function* with *self* bound as the first argument. Allows one to write functions like this:

```
def foo(self, whatever):
    return whatever
```

...outside of the construct of a class.

`gateone.core.utils.minify(path_or_fileobj, kind)`
Returns *path_or_fileobj* as a minified string. *kind* should be one of 'js' or 'css'. Works with JavaScript and CSS files using *slimit* and *cssmin*, respectively.

`gateone.core.utils._minify(path_or_fileobj, kind)`
Returns *path_or_fileobj* as a minified string. *kind* should be one of 'js' or 'css'. Works with JavaScript and CSS files using *slimit* and *cssmin*, respectively.

`gateone.core.utils.get_or_cache(cache_dir, path, minify=True)`
Given a *path*, returns the cached version of that file. If the file has yet to be cached, cache it and return the result. If *minify* is True (the default), the file will be minified as part of the caching process (if possible).

`gateone.core.utils.drop_privileges(uid='nobody', gid='nogroup', supl_groups=None)`
Drop privileges by changing the current process owner/group to *uid/gid* (both may be an integer or a string). If *supl_groups* (list) is given the process will be assigned those values as its effective supplemental groups. If *supl_groups* is None it will default to using 'tty' as the only supplemental group. Example:

```
drop_privileges('gateone', 'gateone', ['tty'])
```


This would change the current process owner to gateone/gateone with 'tty' as its only supplemental group.

Note: On most Unix systems users must belong to the 'tty' group to create new controlling TTYs which is necessary for 'pty.fork()' to work.

Tip: If you get errors like, "OSError: out of pty devices" it likely means that your OS uses something other than 'tty' as the group owner of the devpts filesystem. 'mount | grep pts' will tell you the owner (look for gid=<owner>).

gateone.core.utils.strip_xss(html, whitelist=None, replacement=u'\u2421')

This function returns a tuple containing:

- *html* with all non-whitelisted HTML tags replaced with *replacement*. Any tags that contain JavaScript, VBScript, or other known XSS/executable functions will also be removed.
- A list containing the tags that were removed.

If *whitelist* is not given the following will be used:

```
whitelist = set([
    'a', 'abbr', 'aside', 'audio', 'bdi', 'bdo', 'blockquote', 'canvas',
    'caption', 'code', 'col', 'colgroup', 'data', 'dd', 'del',
    'details', 'div', 'dl', 'dt', 'em', 'figcaption', 'figure', 'h1',
    'h2', 'h3', 'h4', 'h5', 'h6', 'hr', 'i', 'img', 'ins', 'kbd', 'li',
    'mark', 'ol', 'p', 'pre', 'q', 'rp', 'rt', 'ruby', 's', 'samp',
    'small', 'source', 'span', 'strong', 'sub', 'summary', 'sup',
    'time', 'track', 'u', 'ul', 'var', 'video', 'wbr'
])
```

Example:

```
>>> html = '<span>Hello, exploit: </span>'
>>> strip_xss(html)
('<span>Hello, exploit: \u2421</span>', [''])
```

Note: The default *replacement* is the unicode ☐ character (u"\u2421").

If *replacement* is "entities" bad HTML tags will be encoded into HTML entities. This allows things like <script>'whatever'</script> to be displayed without execution (which would be much less annoying to users that were merely trying to share a code example). Here's an example:

```
>>> html = '<span>Hello, exploit: </span>'
>>> strip_xss(html, replacement="entities")
('<span>Hello, exploit: &lt;span&gt;Hello, exploit: &lt;img src="javascript:alert("pwned!")"&gt;&lt;/span&gt;</span>'
 [''])
('<span>Hello, exploit: \u2421</span>', [''])
```

Note: This function should work to protect against all the XSS examples at OWASP. Please let us know if you find something we missed.

gateone.core.utils.create_signature(*parts, **kwargs)

Creates an HMAC signature using the given *parts* and *kwargs*. The first argument **must** be the 'secret' followed by any arguments that are to be part of the hash. The only *kwargs* that is used is 'hmac_algo'. 'hmac_algo' may be any HMAC algorithm present in the hashlib module. If not provided, hashlib.sha1 will be used. Example usage:

```
create_signature(  
    'secret',  
    'some-api-key',  
    'user@somehost',  
    '1234567890123',  
    hmac_algo=hashlib.sha1)
```

Note: The API 'secret' **must** be the first argument. Also, the order *does* matter.

`gateone.core.utils.kill_dtached_proc(session, location, term)`

Kills the dtach processes associated with the given *session* that matches the given *location* and *term*. All the dtach'd sub-processes will be killed as well.

`gateone.core.utils.killall(session_dir, pid_file)`

Kills all running Gate One terminal processes including any detached dtach sessions.

Session_dir The path to Gate One's session directory.

Pid_file The path to Gate One's PID file

1.3.2 JavaScript Code

A large and very important part of Gate One is the client-side JavaScript that runs in the user's browser. This consists of the following:

gateone.js

Gate One's JavaScript ([gateone.js](#)) is made up of several modules (aka plugins), each pertaining to a specific type of activity. These modules are laid out like so:

- **GateOne**
 - `GateOne.Base`
 - `GateOne.Events`
 - `GateOne.i18n`
 - `GateOne.Input` (Source: [gateone_input.js](#))
 - `GateOne.Net`
 - `GateOne.Storage`
 - `GateOne.Visual`
 - `GateOne.User`
 - `GateOne.Utils`

The properties and functions of each respective module are outlined below.

GateOne

The base object for all Gate One modules/plugins.

GateOne.Base

The Base module is mostly copied from [MochiKit](#).

`GateOne.Base.module(parent, name, version[, deps])`

Creates a new *name* module in a *parent* namespace. This function will create a new empty module object with `__name__`, `__version__`, `toString` and `__repr__` properties. It will also verify that all the strings in *deps* are defined in *parent*, or an error will be thrown.

Arguments

- **parent** – The parent module or namespace (object).
- **name** – A string representing the new module name.
- **version** – The version string for this module (e.g. "1.0").
- **deps** – An array of module dependencies, as strings.

The following example would create a new object named, "Net", attach it to the `GateOne` object, at version "1.0", with `GateOne.Base` and `GateOne.Utils` as dependencies:

```
>>> GateOne.Base.module(GateOne, 'Net', '1.0', ['Base', 'Utils']);
>>> GateOne.Net.__repr__();
"[GateOne.Net 1.0]"
>>> GateOne.Net.NAME;
"GateOne.Net"
```

`GateOne.Base.superSandbox(name, dependencies, func)`

A sandbox to wrap JavaScript which will delay-repeat loading itself if *dependencies* are not met. If dependencies cannot be found by the time specified in `GateOne.Base.dependencyTimeout` an exception will be thrown. Here's an example of how to use this function:

```
GateOne.Base.superSandbox("GateOne.ExampleApp", ["GateOne.Terminal"], function(window, undefined) {
    "use strict"; // Don't forget this!

    var stuff = "Put your code here".

});
```

The above example would ensure that `GateOne.Terminal` is loaded before the contents of the `superSandboxed` function are loaded.

Note: Sandboxed functions are always passed the `window` object as the first argument.

You can put whatever globals you like in the dependencies; they don't have to be `GateOne` modules. Here's another example:

```
// Require underscore.js and jQuery:
GateOne.Base.superSandbox("GateOne.ExampleApp", ["_", "jQuery"], function(window, undefined) {
    "use strict";

    var stuff = "Put your code here".

});
```

Name Name of the wrapped function. It will be used to call any `init()` or `postInit()` functions. If you just want dependencies checked you can just pass any unique string.

Dependencies An array of strings containing the JavaScript objects that must be present in the global namespace before we load the contained JavaScript.

Func A function containing the JavaScript code to execute as soon as the dependencies are available.

`GateOne.Base.update(self, obj[, obj2[, objN]])`

Mutate self by replacing its key:value pairs with those from other object(s). Key:value pairs from later objects will overwrite those from earlier objects.

If *self* is null, a new Object instance will be created and returned.

Warning: This mutates *and* returns *self*.

Arguments

- **self** (*object*) – The object you wish to mutate with *obj*.
- **obj** – Any given JavaScript object (e.g. {}).

Returns *self*

GateOne.i18n

A module to store and retrieve localized translations of strings.

`GateOne.i18n.getText(stringOrArray)`

Returns a localized translation of *stringOrArray* if available. If *stringOrArray* is an array it will be joined into a single string via `join('')`.

If no translation of *stringOrArray* is available the text will be returned as-is (or joined, in the case of an Array).

`GateOne.i18n.registerTranslationAction(table)`

Attached to the `go:register_translation` WebSocket action; stores the translation *table* in `GateOne.i18n.translations`.

`GateOne.i18n.setLocales(locales)`

Tells the Gate One server to set the user's locale to *locale*. Example:

```
>>> GateOne.i18n.setLocales(['fr_FR', 'en-US', 'en']);
```

Note: Typically you'd pass `navigator.languages` to this function.

GateOne.prefs

This object holds all of Gate One's preferences. Both those things that are meant to be user-controlled (e.g. theme) and those things that are globally configured (e.g. url). Applications and plugins can store their own preferences here.

GateOne.noSavePrefs

Properties in this object will get ignored when `GateOne.prefs` is saved to `localStorage`

GateOne.Icons

All of Gate One's SVG icons are stored in here (nothing really special about it).

`GateOne.init(prefs[, callback])`

Initializes Gate One using the provided *prefs*. Also performs the initial authentication, performs compatibility checks, and sets up basic preferences.

If *callback* is provided it will be called after `GateOne.Net.connect()` completes.

GateOne.initialize()

Called after `GateOne.init()`, Sets up Gate One's graphical elements (panels and whatnot) and attaches events related to visuals (browser resize and whatnot).

GateOne.Utils

This module consists of a collection of utility functions used throughout Gate One. Think of it like a mini JavaScript library of useful tools.

GateOne.Utills.**init**()

Registers the following WebSocket actions:

- go:save_file -> GateOne.Utills.saveAsAction()
- go:load_style -> GateOne.Utills.loadStyleAction()
- go:load_js -> GateOne.Utills.loadJSAction()
- go:themes_list -> GateOne.Utills.enumerateThemes()

GateOne.Utills.**startBenchmark**()

Put GateOne.Utills.startBenchmark() at the beginning of any code you wish to benchmark (to see how long it takes) and call GateOne.Utills.stopBenchmark() when complete.

GateOne.Utills.**stopBenchmark**([msg])

Put GateOne.Utills.stopBenchmark('optional descriptive message') at the end of any code where you've called GateOne.Utills.startBenchmark().

It will report how long it took to run the code (in the JS console) between startBenchmark() and stopBenchmark() along with a running total of all benchmarks.

GateOne.Utills.**getNode**(nodeOrSelector)

Returns a DOM node if given a [querySelector](#)-style string or an existing DOM node (will return the node as-is).

Note: The benefit of this over just document.querySelector() is that if it is given a node it will return the node as-is (so functions can accept both without having to worry about such things). See removeElement() below for a good example.

Arguments

- **nodeOrSelector** – A [querySelector](#) string like #some_element_id or a DOM node.

Returns A DOM node or null if not found.

Example:

```
>>> var goDivNode = GateOne.Utills.getNode('#gateone'); // Cache it for future lookups
>>> GateOne.Utills.getEmDimensions('#gateone'); // This won't use the cached node
{'w': 8, 'h': 15}
>>> GateOne.Utills.getEmDimensions(goDivNode); // This uses the cached node
{'w': 8, 'h': 15}
```

Both code examples above work because getEmDimensions() uses getNode() to return the node of a given argument. Because of this, getEmDimensions() doesn't require strict string or node arguments (one or the other) and can support both selector strings and nodes at the same time.

GateOne.Utills.**getNodes**(nodeListOrSelector)

Given a CSS [querySelectorAll](#) string (e.g. '.some_class') or [NodeList](#) (in case we're not sure), lookup the node using document.querySelectorAll() and return the result (which will be a [NodeList](#)).

Note: The benefit of this over just document.querySelectorAll() is that if it is given a nodeList it will just return the nodeList as-is (so functions can accept both without having to worry about such things).

Arguments

- **nodeListOrSelector** – A [querySelectorAll](#) string like .some_class or a [NodeList](#).

Returns

A [NodeList](#) or [] (an empty Array) if not found.

Example:

```
>>> var panels = GateOne.Utills.getNodes('#gateone .panel');
```

Note: The *nodeListOrSelector* argument will be returned as-is if it is not a string. It will not actually be checked to ensure it is a proper [NodeList](#).

GateOne.Utills.**partial**(*fn*)

Returns A partially-applied function.

Similar to [MochiKit.Base.partial](#). Returns partially applied function.

Arguments

- **fn** (*function*) – The function to ultimately be executed.
- **arguments** (*arguments*) – Whatever arguments you want to be pre-applied to *fn*.

Example:

```
>>> var addNumbers = function(a, b) {  
    return a + b;  
}  
>>> var addOne = GateOne.Utills.partial(addNumbers, 1);  
>>> addOne(3);  
4
```

Note: This function can also be useful to simply save yourself a lot of typing. If you're planning on calling a function with the same parameters a number of times it is a good idea to use `partial()` to create a new function with all the parameters pre-applied. Can make code easier to read too.

GateOne.Utills.**keys**(*obj*)

Returns an Array containing the keys (attributes) of the given *obj*

GateOne.Utills.**items**(*obj*)

Copied from [MochiKit.Base.items](#). Returns an Array of [propertyName, propertyValue] pairs for the given *obj*.

Arguments

- **obj** (*object*) – Any given JavaScript object.

Returns Array

Example:

```
>>> GateOne.Utills.items(GateOne.terminals).forEach(function(item) { console.log(item) });  
["1", Object]  
["2", Object]
```

Note: Can be very useful for debugging.

GateOne.Utills.**startsWith**(*substr*, *str*)

Returns true if *str* starts with *substr*.

Arguments

- **substr** (*string*) – The string that you want to see if *str* starts with.
- **str** (*string*) – The string you're checking *substr* against.

Returns true/false

Examples:

```
>>> GateOne.Utils.startsWith('some', 'somefile.txt');
true
>>> GateOne.Utils.startsWith('foo', 'somefile.txt');
false
```

GateOne.Utils.**endsWith**(*substr*, *str*)

Returns true if *str* ends with *substr*.

Arguments

- **substr** (*string*) – The string that you want to see if *str* ends with.
- **str** (*string*) – The string you’re checking *substr* against.

Returns true/false

Examples:

```
>>> GateOne.Utils.endsWith('.txt', 'somefile.txt');
true
>>> GateOne.Utils.endsWith('.txt', 'somefile.svg');
false
```

GateOne.Utils.**isArray**(*obj*)

Returns true if *obj* is an Array.

Arguments

- **obj** (*object*) – A JavaScript object.

Returns true/false

Example:

```
>>> GateOne.Utils.isArray(GateOne.terminals['1'].screen);
true
```

GateOne.Utils.**isNodeList**(*obj*)

Returns true if *obj* is a [NodeList](#). [NodeList](#) objects come from DOM level 3 and are what is returned by some browsers when you execute functions like [document.getElementsByTagName](#). This function lets us know if the Array-like object we’ve got is an actual [NodeList](#) (as opposed to an [HTMLCollection](#) or something else like an [Array](#)) or generic object.

Arguments

- **obj** (*object*) – A JavaScript object.

Returns true/false

Example:

```
>>> GateOne.Utils.isNodeList(document.querySelectorAll('.\termline'));
true
```

GateOne.Utils.**isHTMLCollection**(*obj*)

Returns true if *obj* is an [HTMLCollection](#). [HTMLCollection](#) objects come from DOM level 1 and are what is returned by some browsers when you execute functions like [document.getElementsByTagName](#). This function lets us know if the Array-like object we’ve got is an actual [HTMLCollection](#) (as opposed to a [NodeList](#) or just an [Array](#)).

Arguments

- **obj** (*object*) – A JavaScript object.

Returns true/false

Example:

```
>>> GateOne.Utls.isHTMLCollection(document.getElementsByTagName('pre'));
true // Assuming Firefox here
```

Note: The result returned by this function will vary from browser to browser. Sigh.

GateOne.Utls.**isElement**(*obj*)

Returns true if *obj* is an **HTMLElement**.

Arguments

- **obj** (*object*) – A JavaScript object.

Returns true/false

Example:

```
>>> GateOne.Utls.isElement(GateOne.Utls.getNode('#gateone'));
true
```

GateOne.Utls.**removeElement**(*elem*)

Removes the given *elem* from the DOM.

Arguments

- **elem** – A **querySelector** string like `#some_element_id` or a DOM node.

Example:

```
>>> GateOne.Utls.removeElement('#go_infocontainer');
```

GateOne.Utls.**createElement**(*tagname*[, *properties*[, *noprefix*]])

A simplified version of MochiKit's **createDOM** function, it creates a *tagname* (e.g. "div") element using the given *properties*.

Arguments

- **tagname** (*string*) – The type of element to create ("a", "table", "div", etc)
- **properties** (*object*) – An object containing the properties which will be pre-attached to the created element.
- **noprefix** (*boolean*) – If true, will not prefix the created element ID with `GateOne.prefs.prefix`.

Returns A node suitable for adding to the DOM.

Examples:

```
>>> myDiv = GateOne.Utls.createElement('div', {'id': 'foo', 'style': {'opacity': 0.5, 'color': 'black'}})
>>> myAnchor = GateOne.Utls.createElement('a', {'id': 'liftoff', 'href': 'http://liftoffsoftware.com/'});
>>> myParagraph = GateOne.Utls.createElement('p', {'id': 'some_paragraph'});
```

Note: `createElement` will automatically apply `GateOne.prefs.prefix` to the 'id' of the created elements (if an 'id' was given).

GateOne.Utls.**showElement**(*elem*)

Shows the given element (if previously hidden via `hideElement()`) by setting `elem.style.display = 'block'`.

Arguments

- **elem** – A **querySelector** string like `#some_element_id` or a DOM node.

Example:

```
>>> GateOne.Utls.showElement('#go_icon_newterm');
```

GateOne.Utls.**hideElement**(*elem*)

Hides the given element by setting `elem.style.display = 'none'`.

Arguments

- **elem** – A `querySelector` string like `#some_element_id` or a DOM node.

Example:

```
>>> GateOne.Utls.hideElement('#go_icon_newterm');
```

GateOne.Utls.**showElements**(*elems*)

Shows the given elements (if previously hidden via `hideElement()` or `hideElements()`) by setting `elem.style.display = 'block'`.

Arguments

- **elems** – A `querySelectorAll` string like `.some_element_class`, a `NodeList`, or an array.

Example:

```
>>> GateOne.Utls.showElements('.pastearea');
```

GateOne.Utls.**hideElements**(*elems*)

Hides the given elements by setting `elem.style.display = 'none'` on all of them.

Arguments

- **elems** – A `querySelectorAll` string like `.some_element_class`, a `NodeList`, or an array.

Example:

```
>>> GateOne.Utls.hideElements('.pastearea');
```

GateOne.Utls.**getSelText**()

Returns The text that is currently highlighted in the browser.

Example:

```
>>> GateOne.Utls.getSelText();
"localhost" // Assuming the user had highlighted the word, "localhost"
```

GateOne.Utls.**noop**(*a*)

AKA “No Operation”. Returns whatever is given to it (if anything at all). In other words, this function doesn’t do anything and that’s exactly what it is supposed to do!

Arguments

- **a** – Anything you want.

Returns *a*

Example:

```
>>> var functionList = {'1': GateOne.Utls.noop, '2': GateOne.Utls.noop};
```

Note: This function is most useful as a placeholder for when you plan to update *something* in-place later. In the event that *something* never gets replaced, you can be assured that nothing bad will happen if it gets called (no exceptions).

GateOne.Utls.**toArray**(*obj*)

Returns an actual `Array()` given an Array-like *obj* such as an `HTMLCollection` or a `NodeList`.

Arguments

- **obj** (*object*) – An Array-like object.

Returns `Array`

Example:

```
>>> var terms = document.getElementsByClassName(GateOne.prefs.prefix+'terminal');
>>> GateOne.Utills.toArray(terms).forEach(function(termObj) {
    GateOne.Terminal.closeTerminal(termObj.id.split('term')[1]);
});
```

GateOne.Utills.**isEven**(*someNumber*)

Returns true if *someNumber* is even.

Arguments

- **someNumber** (*number*) – A JavaScript object.

Returns true/false

Example:

```
>>> GateOne.Utills.isEven(2);
true
>>> GateOne.Utills.isEven(3);
false
```

GateOne.Utills.**runPostInit**()

Called by GateOne.runPostInit(), iterates over the list of plugins in GateOne.loadedModules calling the init() function of each (if present). When that's done it does the same thing with each respective plugin's postInit() function.

GateOne.Utills.**cacheFileAction**(*fileObj*[, *callback*])

Attached to the 'go:cache_file' WebSocket action; stores the given *fileObj* in the 'fileCache' database and calls *callback* when complete.

If *fileObj['kind']* is 'html' the file will be stored in the 'html' table otherwise the file will be stored in the 'other' table.

GateOne.Utills.**loadJSAction**(*message*)

Loads a JavaScript file sent via the go:load_js WebSocket action into a <script> tag inside of GateOne.prefs.goDiv (not that it matters where it goes).

If *message.cache* is false or *noCache* is true, will not update the fileCache database with this incoming file.

GateOne.Utills.**loadStyleAction**(*message*[, *noCache*])

Loads the stylesheet sent via the go:load_style WebSocket action. The *message* is expected to be a JSON object that contains the following objects:

result Must be "Success" if delivering actual CSS. Anything else will be reported as an error in the JS console.

css Must be true.

data The actual stylesheet (the CSS).

cache If false the stylesheet will not be cached at the client (stored in the fileCache database).

media *Optional*: If provided this value will be used as the "media" attribute inside the created <style> tag.

Example message object:

```
{
  "result": "Success",
  "css": true,
  "data": ".someclass:hover {cursor: pointer;}",
  "media": "screen",
  "cache": true
}
```

If called directly (as opposed to via the WebSocket action) the *noCache*

`GateOne.Utills.loadTheme(theme)`

Sends the `go:get_theme` WebSocket action to the server asking it to send/sync/load the given *theme*.

Arguments

- **theme** (*string*) – The theme you wish to load.

Example:

```
>>> GateOne.Utills.loadTheme("white");
```

`GateOne.Utills.enumerateThemes(messageObj)`

Attached to the `go:themes_list` WebSocket action; updates the preferences panel with the list of themes stored on the server.

`GateOne.Utills.savePrefs(skipNotification)`

Saves what's set in `GateOne.prefs` to `localStorage[GateOne.prefs.prefix+'prefs']` as JSON; skipping anything that's set in `GateOne.noSavePrefs`.

Displays a notification to the user that preferences have been saved.

Arguments

- **skipNotification** (*boolean*) – If true, don't notify the user that prefs were just saved.

`GateOne.Utills.loadPrefs`

Populates `GateOne.prefs` with values from `localStorage[GateOne.prefs.prefix+'prefs']`.

`GateOne.Utills.xhrGet(url[, callback])`

Performs a GET on the given *url* and if given, calls *callback* with the *responseText* as the only argument.

Arguments

- **url** (*string*) – The URL to GET.
- **callback** (*function*) – A function to call like so: `callback(responseText)`

Example:

```
>>> var mycallback = function(responseText) { console.log("It worked: " + responseText) };
>>> GateOne.Utills.xhrGet('https://demo.example.com/static/about.html', mycallback);
It worked: <!DOCTYPE html>
<html>
<head>
...

```

`GateOne.Utills.isVisible(elem)`

Returns true if *node* is visible (checks parent nodes recursively too). *node* may be a DOM node or a selector string.

Example:

```
>>> GateOne.Utills.isVisible('#'+GateOne.prefs.prefix+'pastearea1');
true

```

Note: Relies on checking `elem.style.opacity` and `elem.style.display`. Does *not* check transforms.

`GateOne.Utills.getCookie(name)`

Returns the given cookie (*name*).

Arguments

- **name** (*string*) – The name of the cookie to retrieve.

Examples:

```
>>> GateOne.Utls.getCookie(GateOne.prefs.prefix + 'gateone_user'); // Returns the 'gateone_user' cookie
```

GateOne.Utls.**setCookie**(*name, value, days*)

Sets the cookie of the given *name* to the given *value* with the given number of expiration *days*.

Arguments

- **name** (*string*) – The name of the cookie to retrieve.
- **value** (*string*) – The value to set.
- **days** (*number*) – The number of days the cookie will be allowed to last before expiring.

Examples:

```
>>> GateOne.Utls.setCookie('test', 'some value', 30); // Sets the 'test' cookie to 'some value' with an expiration of 30 days
```

GateOne.Utls.**deleteCookie**(*name, path, domain*)

Deletes the given cookie (*name*) from *path* for the given *domain*.

Arguments

- **name** (*string*) – The name of the cookie to delete.
- **path** (*string*) – The path of the cookie to delete (typically '/' but could be '/some/path/on/the/webserver' =).
- **path** – The domain where this cookie is from (an empty string means "the current domain in window.location.href").

Example:

```
>>> GateOne.Utls.deleteCookie('gateone_user', '/', ''); // Deletes the 'gateone_user' cookie
```

GateOne.Utls.**randomString**(*length*[, *chars*])

Returns A random string of the given *length* using the given *chars*.

If *chars* is omitted the returned string will consist of lower-case ASCII alphanumerics.

Arguments

- **length** (*int*) – The length of the random string to be returned.
- **chars** (*string*) – *Optional*: a string containing the characters to use when generating the random string.

Example:

```
>>> GateOne.Utls.randomString(8);
"oa2f9txf"
>>> GateOne.Utls.randomString(8, '123abc');
"1b3ac12b"
```

GateOne.Utls.**saveAs**(*blob, filename*)

Saves the given *blob* (which must be a proper [Blob](#) object with data inside of it) as *filename* (as a file) in the browser. Just as if you clicked on a link to download it.

Note: This is amazingly handy for downloading files over the [WebSocket](#).

GateOne.Utls.**saveAsAction**(*message*)

Note: This function is attached to the 'save_file' [WebSocket](#) action (in `GateOne.Net.actions`) via `GateOne.Utls.init()`.

Saves to disk the file contained in *message*. The *message* object should contain the following:

- result** Either 'Success' or a descriptive error message.
- filename** The name we'll give to the file when we save it.
- data** The content of the file we're saving.
- mimetype** *Optional:* The mimetype we'll be instructing the browser to associate with the file (so it will handle it appropriately). Will default to 'text/plain' if not given.

GateOne.Utills.**isPageHidden()**

Returns true if the page (browser tab) is hidden (e.g. inactive). Returns false otherwise.

Example:

```
>>> GateOne.Utills.isPageHidden();
false
```

GateOne.Utills.**createBlob(array, mimetype)**

Returns a Blob() object using the given *array* and *mimetype*. If *mimetype* is omitted it will default to 'text/plain'. Optionally, *array* may be given as a string in which case it will be automatically wrapped in an array.

Arguments

- **array** (*array*) – A string or array containing the data that the Blob will contain.
- **mimetype** (*string*) – A string representing the mimetype of the data (e.g. 'application/javascript').

Returns A Blob()

Note: The point of this function is favor the Blob() function while maintaining backwards-compatibility with the deprecated BlobBuilder interface (for browsers that don't support Blob() yet).

Example:

```
>>> var blob = GateOne.Utills.createBlob('some data here', 'text/plain');
```

GateOne.Utills.**getQueryVariable(variable[, url])**

Returns the value of a query string variable from window.location.

If no matching variable is found, returns undefined. Example:

```
>>> // Assume window.location.href = 'https://gateone/?foo=bar,bar,bar'
>>> GateOne.Utills.getQueryVariable('foo');
'bar,bar,bar'
```

Optionally, a *url* may be specified to perform the same evaluation on *url* instead of window.location.

GateOne.Utills.**removeQueryVariable(variable)**

Removes the given query string variable from window.location.href using window.history.replaceState(). Leaving all other query string variables alone.

Returns the new query string.

GateOne.Utills.**insertAfter(newElement, targetElement)**

The opposite of the DOM's built in insertBefore() function; inserts the given *newElement* after *targetElement*.

targetElement may be given as a pre-constructed node object or a querySelector-like string.

GateOne.Utils.**debounce**(*func, wait, immediate*)

A copy of [the debounce function](#) from the excellent underscore.js.

GateOne.Logging

Gate One's Logging module provides functions for logging to the console (or whatever destination you like) and supports multiple log levels:

Level	Name	Default Console Function
10	DEBUG	console.debug()
20	INFO	console.log()
30	WARNING	console.warn()
40	ERROR	console.error()
50	FATAL	console.error()

If a particular console function is unavailable the `console.log()` function will be used as a fallback.

Tip: You can add your own destinations; whatever you like! See the `GateOne.Logging.addDestination()` function for details.

Shortcuts:

There are various shortcut functions available to save some typing:

- `GateOne.Logging.logDebug()`
- `GateOne.Logging.logInfo()`
- `GateOne.Logging.logWarning()`
- `GateOne.Logging.logError()`
- `GateOne.Logging.logFatal()`

It is recommended that you assign these shortcuts at the top of your code like so:

```
var logFatal = GateOne.Logging.logFatal,  
    logError = GateOne.Logging.logError,  
    logWarning = GateOne.Logging.logWarning,  
    logInfo = GateOne.Logging.logInfo,  
    logDebug = GateOne.Logging.logDebug;
```

That way you can just add "`logDebug()`" anywhere in your code and it will get logged appropriately to the default destinations (with a nice timestamp and whatnot).

GateOne.Logging.init()

Initializes logging by setting `GateOne.Logging.level` using the value provided by `GateOne.prefs.logLevel`. `GateOne.prefs.logLevel` may be given as a case-insensitive string or an integer.

Also, if `GateOne.prefs.logToServer` is false `GateOne.Logging.logToConsole()` will be removed from `GateOne.Logging.destinations`.

GateOne.Logging.setLevel(level)

Sets the log *level* to an integer if the given a string (e.g. "DEBUG"). Sets it as-is if it's already a number. Examples:

```
>>> GateOne.Logging.setLevel(10); // Set log level to DEBUG  
>>> GateOne.Logging.setLevel("debug") // Same thing; they both work!
```

GateOne.Logging.**log**(*msg*[, *level*[, *destination*]])

Logs the given *msg* using all of the functions in GateOne.Logging.destinations after being prepended with the date and a string indicating the log level (e.g. "692011-10-25 10:04:28 INFO <msg>") if *level* is determined to be greater than the value of GateOne.Logging.level. If the given *level* is not greater than GateOne.Logging.level *msg* will be discarded (noop).

level can be provided as a string, an integer, null, or be left undefined:

- If an integer, an attempt will be made to convert it to a string using GateOne.Logging.levels but if this fails it will use "lvl:<integer>" as the level string.
- If a string, an attempt will be made to obtain an integer value using GateOne.Logging.levels otherwise GateOne.Logging.level will be used (to determine whether or not the message should actually be logged).
- If undefined, the level will be set to GateOne.Logging.level.
- If null (as opposed to undefined), level info will not be included in the log message.

If *destination* is given (must be a function) it will be used to log messages like so: destination(message, levelStr). The usual conversion of *msg* to *message* will apply.

Any additional arguments after *destination* will be passed directly to that function.

GateOne.Logging.**logToConsole**(*msg*, *level*)

Logs the given *msg* to the browser's JavaScript console. If *level* is provided it will attempt to use the appropriate console logger (e.g. console.warn()).

Note: The original version of this function is from: MochiKit.Logger.prototype.logToConsole.

GateOne.Logging.**logToServer**(*msg*[, *level*])

Sends the given log *msg* to the Gate One server. Such messages will end up in 'logs/gateone-client.log'.

GateOne.Logging.**addDestination**(*name*, *dest*)

Creates a new log destination named, *name* that calls function *dest* like so:

```
>>> dest(message);
```

Example usage:

```
>>> GateOne.Logging.addDestination('screen', GateOne.Visual.displayMessage);
```

Note: The above example is kind of fun. Try it in your JavaScript console!

Tip: With the right function you can send client log messages *anywhere*.

GateOne.Logging.**removeDestination**(*name*)

Removes the given log destination (*name*) from GateOne.Logging.destinations

GateOne.Logging.**dateFormatter**(*dateObj*)

Converts a Date() object into string suitable for logging. Example:

```
>>> GateOne.Logging.dateFormatter(new Date());
"2013-08-15 08:45:41"
```

GateOne.Net

Just about all of Gate One's communications with the server are handled inside this module. It contains all the functions and properties to deal with setting up the [WebSocket](#)

and issuing/receiving commands over it. The most important facet of GateOne.Net is GateOne.Net.actions which holds the mapping of what function maps to which command. More info on GateOne.Net.actions is below.

GateOne.Net.actions

This is where all of Gate One's [WebSocket](#) protocol actions are assigned to functions. Here's how they are defined by default:

Action	Function
go:gateone_user	GateOne.User.storeSessionAction()
go:load_css	GateOne.Visual.CSSPluginAction()
go:load_style	GateOne.Utils.loadStyleAction()
go:log	GateOne.Net.log()
go:notice	GateOne.Visual.serverMessageAction()
go:user_message	GateOne.Visual.userMessageAction()
go:ping	GateOne.Net.ping()
go:pong	GateOne.Net.pong()
go:reauthenticate	GateOne.Net.reauthenticate()
go:save_file	GateOne.Utils.saveAsAction()
go:set_username	GateOne.User.setUsernameAction()
go:timeout	GateOne.Terminal.timeoutAction()

Note: Most of the above is added via addAction() inside of each respective module's init() function.

For example, if we execute GateOne.Net.ping(), this will send a message over the [WebSocket](#) like so:

```
GateOne.ws.send(JSON.stringify({'ping': timestamp}));
```

The GateOne server will receive this message and respond with a pong message that looks like this (Note: Python code below):

```
message = {'pong': timestamp} # The very same timestamp we just sent via GateOne.Net.ping()
self.write_message(json_encode(message))
```

When GateOne.Net receives a message from the server over the [WebSocket](#) it will evaluate the object it receives as {action: message} and call the matching action in GateOne.Net.actions. In this case, our action "pong" matches GateOne.Net.actions['pong'] so it will be called like so:

```
GateOne.Net.actions['pong'](message);
```

Plugin authors can add their own arbitrary actions using GateOne.Net.addAction(). Here's an example taken from the SSH plugin:

```
GateOne.Net.addAction('sshjs_connect', GateOne.SSH.handleConnect);
GateOne.Net.addAction('sshjs_reconnect', GateOne.SSH.handleReconnect);
```

If no action can be found for a message it will be passed to GateOne.Visual.displayMessage() and displayed to the user like so:

```
GateOne.Visual.displayMessage('Message From Server: ' + <message>);
```

GateOne.Net.init()

Assigns the go:ping_timeout event (which just displays a message to the user indicating as such).

GateOne.Net.**sendChars**()

Deprecated since version 1.2: Use GateOne.Terminal.sendChars() instead.

GateOne.Net.**sendString**()

Deprecated since version 1.2: Use GateOne.Terminal.sendString() instead.

GateOne.Net.**log**(*message*)

Arguments

- **message** (*string*) – The message received from the Gate One server.

This function can be used in debugging GateOne.Net.actions; it logs whatever message is received from the Gate One server: GateOne.Logging.logInfo(message) (which would equate to console.log under most circumstances).

When developing a new action, you can test out or debug your server-side messages by attaching the respective action to GateOne.Net.log() like so:

```
GateOne.Net.addAction('my_action', GateOne.Net.log);
```

Then you can view the exact messages received by the client in the JavaScript console in your browser.

Tip: Executing GateOne.Logging.setLevel('DEBUG') in your JS console will also log all incoming messages from the server (though it can be a bit noisy).

GateOne.Net.**ping**([*logLatency*])

Sends a 'ping' to the server over the WebSocket. The response from the server is handled by GateOne.Net.pong().

If a response is not received within a certain amount of time (milliseconds, controlled via GateOne.prefs.pingTimeout) the WebSocket will be closed and a go:ping_timeout event will be triggered.

If *logLatency* is true (the default) the latency will be logged to the JavaScript console via GateOne.Logging.logInfo().

Note: The default value for GateOne.prefs.pingTimeout is 5 seconds. You can change this setting via the js_init option like so: --js_init='{pingTimeout: "5000"}' (command line) or in your 10server.conf ("js_init": "{pingTimeout: '5000'}").

GateOne.Net.**pong**(*timestamp*)

Arguments

- **timestamp** (*string*) – Expected to be the output of new Date().toISOString() (as generated by ping()).

Simply logs *timestamp* using GateOne.Logging.logInfo() and includes a measurement of the round-trip time in milliseconds.

GateOne.Net.**reauthenticate**()

Called when the Gate One server wants us to re-authenticate our session (e.g. our cookie expired). Deletes the 'gateone_user' cookie and reloads the current page.

This will force the client to re-authenticate with the Gate One server.

To disable the automatic reload set GateOne.Net.reauthForceReload = false.

GateOne.Net.**sendDimensions**()

Deprecated since version 1.2: Use GateOne.Terminal.sendDimensions() instead.

GateOne.Net.**blacklisted**(*msg*)

Called when the server tells us the client has been blacklisted (i.e for abuse). Sets

GateOne.Net.connect = GateOne.Utills.noop; so a new connection won't be attempted after being disconnected. It also displays a message to the user from the server.

GateOne.Net.**connectionError**(*msg*)

Called when there's an error communicating over the [WebSocket](#)... Displays a message to the user indicating there's a problem, logs the error (using `logError()`), and sets a five-second timeout to attempt reconnecting.

This function is attached to the WebSocket's `onclose` event and shouldn't be called directly.

GateOne.Net.**sslError**()

Called when we fail to connect due to an SSL error (user must accept the SSL certificate). It displays a message to the user that gives them the option to open up a new page where they can accept the SSL certificate (it automatically redirects them back to the current page).

GateOne.Net.**connect**(*[callback]*)

Opens a connection to the [WebSocket](#) defined in `GateOne.prefs.url` and stores it as `GateOne.ws`. Once connected `GateOne.initialize()` will be called.

If an error is encountered while trying to connect to the [WebSocket](#), `GateOne.Net.connectionError()` will be called to notify the user as such. After five seconds, if a connection has yet to be connected successfully it will be assumed that the user needs to accept the Gate One server's SSL certificate. This will invoke call to `GateOne.Net.sslError()` which will redirect the user to the `accept_certificate.html` page on the Gate One server. Once that page has loaded successfully (after the user has clicked through the interstitial page) the user will be redirected back to the page they were viewing that contained Gate One.

Note: This function gets called by `GateOne.init()` and there's really no reason why it should be called directly by anything else.

GateOne.Net.**onClose**(*evt*)

Attached to `GateOne.ws.onclose()`; called when the WebSocket is closed.

If `GateOne.Net.connectionProblem` is true `GateOne.Net.connectionError()` will be called.

GateOne.Net.**disconnect**(*[reason]*)

Closes the WebSocket and clears all processes (timeouts/keepalives) that watch the state of the connection.

If a *reason* is given it will be passed to the WebSocket's `close()` function as the only argument.

Note: The *reason* feature of WebSockets does not appear to be implemented in any browsers (yet).

GateOne.Net.**onOpen**(*[callback]*)

This gets attached to `GateOne.ws.onopen` inside of `connect()`. It clears any error message that might be displayed to the user and asks the server to send us the (currently-selected) theme CSS and all plugin JS/CSS. It then sends an authentication message (the `go:authenticate` WebSocket action) and calls `GateOne.Net.ping()` after a short timeout (to let things settle down lest they interfere with the ping time calculation).

Lastly, it fires the `go:connection_established` event.

GateOne.Net.**onMessage**(*evt*)

Arguments

- **event** (*event*) – A `WebSocket` event object as passed by the 'message' event.

This gets attached to `GateOne.ws.onmessage` inside of `connect()`. It takes care of decoding (`JSON`) messages sent from the server and calling any matching actions. If no matching action can be found inside `event.data` it will fall back to passing the message directly to `GateOne.Visual.displayMessage()`.

`GateOne.Net.timeoutAction()`

Writes a message to the screen indicating a session timeout has occurred (on the server) and closes the `WebSocket`.

`GateOne.Net.addAction(name, func)`

Arguments

- **name** (*string*) – The name of the action we're going to attach *func* to.
- **func** (*function*) – The function to be called when an action arrives over the `WebSocket` matching *name*.

Adds an action to the `GateOne.Net.actions` object.

Example:

```
>>> GateOne.Net.addAction('sshjs_connect', GateOne.SSH.handleConnect);
```

`GateOne.Net.setTerminal()`

Deprecated since version 1.2: Use `GateOne.Terminal.setTerminal()` instead.

`GateOne.Net.killTerminal()`

Deprecated since version 1.2: Use `GateOne.Terminal.killTerminal()` instead.

`GateOne.Net.refresh()`

Deprecated since version 1.2: Use `GateOne.Terminal.refresh()` instead.

`GateOne.Net.fullRefresh()`

Deprecated since version 1.2: Use `GateOne.Terminal.fullRefresh()` instead.

`GateOne.Net.getLocations()`

Asks the server to send us a list of locations via the `go:get_locations` `WebSocket` action. Literally:

```
>>> GateOne.ws.send(JSON.stringify({'go:get_locations': null}));
```

This will ultimately result in `GateOne.Net.locationsAction()` being called.

`GateOne.Net.locationsAction()`

Attached to the `go:locations` `WebSocket` action. Sets `GateOne.locations` to *locations* which should be an object that looks something like this:

```
{ "default":
  { "terminal": {
    "1": {
      "created": 1380590438000,
      "command": "SSH",
      "title": "user@enterprise: ~"
    },
    "2": {
      "created": 1380590633000,
      "command": "login",
      "title": "root@enterprise: /var/log"
    },
    "x11": {
      "1": {
```

```
        "created":1380590132000,
        "command":"google-chrome-unstable",
        "title":"Liftoff Software | Next stop, innovation - Google Chrome"
    },
    "2":{
        "created":1380591192000,
        "command":"subl",
        "title":"~/workspace/SuperSandbox/SuperSandbox.js - Sublime Text (UNREGISTERED)"
    },
    },
    "transfer":{
        "1":{
            "created":1380590132000,
            "command":"Unknown",
            "title":"From: bittorrent://kubuntu-13.04-desktop-armhf+omap4.img.torrent To: sftp://user@ente
        },
    }
}
```

GateOne.Net.**setLocation**(*location*)

Arguments

- **location** (*string*) – A string containing no spaces.

Sets GateOne.location to *location* and sends a message (the go:set_location WebSocket action) to the Gate One server telling it to change the current location to *location*.

GateOne.**Visual**

This module contains all of Gate One's visual effect functions. It is just like GateOne.Utils but specific to visual effects and DOM manipulations.

GateOne.Visual.**goDimensions**

Stores the dimensions of the GateOne.prefs.goDiv element in the form of {w: '800', h: '600'} where 'w' and 'h' represent the width and height in pixels. It is used by several functions in order to calculate how far to slide terminals, how many rows and columns will fit, etc.

Registers the following WebSocket actions:

Action	Function
go:notice	GateOne.Visual.serverMessageAction()
go:user_message	GateOne.Visual.userMessageAction()

GateOne.Visual.**init**()

Adds the 'grid' icon to the toolbar for users to click on to bring up/down the grid view.

Registers the following Gate One events:

Event	Function
go:switch_workspace	GateOne.Visual.slideToWorkspace()
go:switch_workspace	GateOne.Visual.locationsCheck()
go:cleanup_workspaces	GateOne.Visual.cleanupWorkspaces()

Registers the following DOM events:

Element	Event	Function
window	resize	GateOne.Visual.updateDimensions()

GateOne.Visual.**postInit**()

Sets up our default keyboard shortcuts and opens the application chooser if no other applications have opened themselves after a short timeout (500ms).

Registers the following keyboard shortcuts:

Function	Shortcut
New Workspace	Control-Alt-N
Close Workspace	Control-Alt-W
Show Grid	Control-Alt-G
Switch to the workspace on the left	Shift-LeftArrow
Switch to the workspace on the right	Shift-RightArrow
Switch to the workspace above	Shift-UpArrow
Switch to the workspace below	Shift-DownArrow

`GateOne.Visual.locationsPanel()`

Creates the locations panel and adds it to `GateOne.node` (hidden by default).

`GateOne.Visual.showLocationsIcon()`

Creates then adds the location panel icon to the toolbar.

`GateOne.Visual.showLocationsIcon()`

Removes the locations panel icon from the toolbar.

`GateOne.Visual.locationsCheck(workspaceNum)`

Will add or remove the locations panel icon to/from the toolbar if the application residing in the current workspace supports locations.

`GateOne.Visual.appChooser([where])`

Creates a new application chooser (akin to a browser's "new tab tab") that displays the application selection screen (and possibly other things in the future).

If *where* is undefined a new workspace will be created and the application chooser will be placed there. If *where* is false the new application chooser element will be returned without placing it anywhere.

`GateOne.Visual.setTitle(title)`

Sets the innerHTML of the 'sideinfo' element to *title*.

Note: The location of the 'sideinfo' is controlled by the theme but it is typically on the right-hand side of the window.

`GateOne.Visual.updateDimensions()`

Sets `GateOne.Visual.goDimensions` to the current width/height of `GateOne.prefs.goDiv`. Typically called when the browser window is resized.

```
>>> GateOne.Visual.updateDimensions();
```

Also sends the "go:set_dimensions" WebSocket action to the server so that it has a reference of the client's width/height as well as information about the size of the `goDiv` (usually `#gateone`) element and the size of workspaces.

`GateOne.Visual.transitionEvent()`

Returns the correct name of the 'transitionend' event for the current browser. Example:

```
>>> console.log(GateOne.Visual.transitionEvent()); // Pretend we're using Chrome
'webkitTransitionEnd'
>>> console.log(GateOne.Visual.transitionEvent()); // Pretend we're using Firefox
'transitionend'
```

`GateOne.Visual.applyTransform(obj, transform[, callback1[, callbackN]])`

Arguments

- **obj** – A `querySelector` string like `#some_element_id`, a DOM node, an `Array` of DOM nodes, an `HTMLCollection`, or a `NodeList`.

- **transform** – A [CSS3 transform](#) function such as `scale()` or `translate()`.
- **callbacks** – Any number of functions can be supplied to be called back after the transform is applied. Each callback will be called after the previous one has completed. This allows the callbacks to be chained one after the other to create animations. (see below)

This function is Gate One's bread and butter: It applies the given CSS3 *transform* to *obj*. *obj* can be one of the following:

- A [querySelector](#) -like string (e.g. `"#some_element_id"`).
- A DOM node.
- An [Array](#) or an Array-like object containing DOM nodes such as [HTMLCollection](#) or [NodeList](#) (it will apply the transform to all of them).

The *transform* should be *just* the actual transform function (e.g. `scale(0.5)`). `applyTransform()` will take care of applying the transform according to how each browser implements it. For example:

```
>>> GateOne.Visual.applyTransform('#somediv', 'translateX(500%)');
```

...would result in `#somediv` getting styles applied to it like this:

```
#somediv {
  -webkit-transform: translateX(500%); // Chrome/Safari/Webkit-based stuff
  -moz-transform: translateX(500%);   // Mozilla/Firefox/Gecko-based stuff
  -o-transform: translateX(500%);     // Opera
  -ms-transform: translateX(500%);    // IE9+
  -khtml-transform: translateX(500%); // Konqueror
  transform: translateX(500%);        // Some day this will be all that is necessary
}
```

Optionally, any amount of callback functions may be provided which will be called after each transform (aka transition) completes. These callbacks will be called in a chain with the next callback being called after the previous one is complete. Example:

```
>>> // Chain three moves of #gateone; each waiting for the previous transition to complete before continuing
>>> GateOne.Visual.applyTransform(GateOne.node, 'translateX(-2%)', function() { GateOne.Visual.applyTransf
```

`GateOne.Visual.applyStyle(elem, style)`

Arguments

- **elem** – A [querySelector](#) string like `#some_element_id` or a DOM node.
- **style** – A JavaScript object holding the style that will be applied to *elem*.

A convenience function that allows us to apply multiple style changes in one go. For example:

```
>>> GateOne.Visual.applyStyle('#somediv', {'opacity': 0.5, 'color': 'black'});
```

`GateOne.Visual.getTransform(elem)`

Arguments

- **elem** (*number*) – A [querySelector](#) string ID or a DOM node.

Returns the transform string applied to the style of the given *elem*

```
>>> GateOne.Visual.getTransform('#go_term1_pre');
"translateY(-3px)"
```

`GateOne.Visual.togglePanel(panel[, callback])`

Toggles the given *panel* in or out of view. If other panels are open at the time, they will be closed. If *panel* evaluates to false, all open panels will be closed.

This function also has some events that can be hooked into:

- When the panel is toggled out of view: `GateOne.Events.trigger("go:panel_toggle:out", panelElement)`
- When the panel is toggled into view: `GateOne.Events.trigger("go:panel_toggle:in", panelElement)`

You can hook into these events like so:

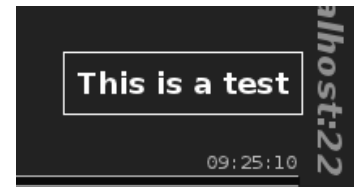
```
>>> GateOne.Events.on("go:panel_toggle:in", myFunc); // When panel is toggled into view
>>> GateOne.Events.on("go:panel_toggle:out", myFunc); // When panel is toggled out of view
```

If a *callback* is given it will be called after the panel has *completed* being toggled *in* (i.e. after animations have completed).

```
GateOne.Visual.displayMessage(message[, timeout[, removeTimeout[, id]]])
```

Arguments

- **message** (*string*) – The message to display.
- **timeout** (*integer*) – Milliseconds; How long to display the message before starting the *removeTimeout* timer. **Default:** 1000.
- **removeTimeout** (*integer*) – Milliseconds; How long to delay before calling `GateOne.Utils.removeElement()` on the message DIV. **Default:** 5000.
- **id** (*string*) – The ID to assign the message DIV. **Default:** `GateOne.prefs.prefix+"notice"`.
- **noLog** (*boolean*) – If set to true the message will not be logged.



Displays *message* to the user via a transient pop-up DIV that will appear inside `GateOne.prefs.goDiv`. How long the message lasts can be controlled via *timeout* and *removeTimeout* (which default to 1000 and 5000, respectively).

If *id* is given, it will be prefixed with `GateOne.prefs.prefix` and used as the DIV ID for the pop-up. i.e. `GateOne.prefs.prefix+id`. The default is `GateOne.prefs.prefix+"notice"`.

```
>>> GateOne.Visual.displayMessage('This is a test.');
```

Note: The default is to display the message in the lower-right corner of `GateOne.prefs.goDiv` but this can be controlled via CSS.

```
GateOne.Visual.handleVisibility(e)
```

This function gets called whenever a tab connected to Gate One becomes visible or invisible. Triggers the `go:visible` and `go:invisible` events.

```
GateOne.Visual.newWorkspace()
```

Creates a new workspace on the grid and returns the DOM node that is the new workspace.

If the currently-selected workspace happens to be the application chooser it will be emptied and returned instead of creating a new one.

```
GateOne.Visual.closeWorkspace(workspace)
```

Removes the given *workspace* from the 'gridwrapper' element and triggers the `go:close_workspace` event.

If *message* (string) is given it will be displayed to the user when the workspace is closed.

Note: If you're writing an application for Gate One you'll definitely want to attach a function to the `go:close_workspace` event to close your application.

`GateOne.Visual.switchWorkspace(workspace)`

Triggers the `go:switch_workspace` event which by default calls `GateOne.Visual.slideToWorkspace()`.

Tip: If you wish to use your own workspace-switching animation just write your own function to handle it and call `GateOne.Events.off('go:switch_workspace', GateOne.Visual.slideToWorkspace); GateOne.Events.on('go:switch_workspace', your-Function);`

`GateOne.Visual.cleanupWorkspaces()`

This gets attached to the `'go:cleanup_workspaces'` event which should be triggered by any function that may leave a workspace empty. It walks through all the workspaces and removes any that are empty.

For example, let's say your app just removed itself from the workspace as a result of a server-controlled action (perhaps a BOFH killed the user's process). At the end of your `closeMyApp()` function you want to put this:

```
GateOne.Events.trigger("go:cleanup_workspaces");
```

Note: Make sure you trigger the event instead of calling this function directly so that other attached functions can do their part.

Why is this mechanism the opposite of everything else where you call the function and that function triggers its associated event? Embedded mode, of course! In embedded mode the parent web page may use something other than workspaces (e.g. tabs). In embedded mode this function never gets attached to the `go:cleanup_workspaces` event so this function will never get called. This allows the page embedding Gate One to attach its own function to this event to perform an equivalent action (for whatever workspace-like mechanism it is using).

`GateOne.Visual.relocateWorkspace(workspace, location)`

Relocates the given *workspace* (number) to the given *location* by firing the `go:relocate_workspace` event and *then* closing the workspace (if not already closed). The given *workspace* and *location* will be passed to the event as the only arguments.

The `'data-application'` attribute of the DOM node associated with the given *workspace* will be used to determine whether or not the application running on the workspace is relocatable. It does this by checking the matching application's `'__appinfo__.relocatable'` attribute.

Applications that support relocation must ensure that they set the appropriate `'data-application'` attribute on the workspace if they create workspaces on their own.

`GateOne.Visual.slideToWorkspace(workspace)`

Slides the view to the given *workspace*. If `GateOne.Visual.noReset` is true, don't reset the grid before switching.

`GateOne.Visual.stopIndicator(direction)`

Displays a visual indicator (appearance determined by theme) that the user cannot slide in given *direction*. Example:


```
>>> GateOne.Visual.stopIndicator('left');
```

The given *direction* may be one of: **left, right, up, down**.

GateOne.Visual.**slideLeft**()

Slides to the workspace left of the current view.

GateOne.Visual.**slideRight**()

Slides to the workspace right of the current view.

GateOne.Visual.**slideDown**()

Slides the view downward one workspace by pushing all the others up.

GateOne.Visual.**slideUp**()

Slides the view downward one workspace by pushing all the others down.

GateOne.Visual.**resetGrid**(*animate*)

Places all workspaces in their proper position in the grid. By default this happens instantly with no animations but if *animate* is true CSS3 transitions will take effect.

GateOne.Visual.**gridWorkspaceDragStart**(*e*)

Called when the user starts dragging a workspace in grid view; creates drop targets above each workspace and sets up the 'dragover', 'dragleave', and 'drop' events.

This function is also responsible for creating the thumbnail of the workspace being dragged.

GateOne.Visual.**gridWorkspaceDragOver**(*e*)

Attached to the various drop targets while a workspace is being dragged in grid view; sets the style of the drop target to indicate to the user that the workspace can be dropped there.

GateOne.Visual.**gridWorkspaceDragLeave**(*e*)

Attached to the various drop targets while a workspace is being dragged in grid view; sets the background color of the drop target back to 'transparent' to give the user a clear visual indication that the drag is no longer above the drop target.

GateOne.Visual.**gridWorkspaceDrop**(*e*)

Attached to the various drop targets while a workspace is being dragged in grid view; handles the 'drop' of a workspace on to another. Will swap the dragged workspace with the one to which it was dropped by calling GateOne.Visual.swapWorkspaces()

GateOne.Visual.**swapWorkspaces**(*ws1, ws2*)

Swaps the location of the given workspaces in the grid and fires the go:swapped_workspaces event with *ws1* and *ws2* as the arguments.

Ws1 number The workspace number.

Ws2 number The other workspace number.

GateOne.Visual.**toggleGridView**(*[goBack]*)

Brings up the workspace grid view or returns to full-size.

If *goBack* is false, don't bother switching back to the previously-selected workspace

GateOne.Visual.**createGrid**(*id, workspaceNames*)

Creates a container for all the workspaces and optionally pre-creates workspaces using *workspaceNames*.

id will be the ID of the resulting grid (e.g. "gridwrapper").

workspaceNames is expected to be a list of DOM IDs.

GateOne.Visual.**serverMessageAction**(*message*)

Attached to the go:notice WebSocket action; displays a given *message* from the Gate One server as a transient pop-up using GateOne.Visual.displayMessage().

GateOne.Visual.**userMessageAction**(*message*)

Attached to the go:user_message WebSocket action; displays a given *message* as a transient pop-up using GateOne.Visual.displayMessage().

Note: This will likely change to include/use additional metadata in the future (such as from, to, etc)

GateOne.Visual.**dialog**(*title*, *content*[, *options*])

Creates an in-page dialog with the given *title* and *content*. Returns a function that will close the dialog when called.

Dialogs can be moved around and closed at-will by the user with a clearly visible title bar that is always present.

All dialogs are placed within the GateOne.prefs.goDiv container but have their position set to 'fixed' so they can be moved anywhere on the page (even outside of the container where Gate One resides).

Arguments

- **title** (*string*) – Will appear at the top of the dialog.
- **content** (*stringOrNode*) – String or JavaScript DOM node - The content of the dialog.
- **options** (*object*) – An associative array of parameters that change the look and/or behavior of the dialog. See below.

Options

events An object containing DOM events that will be attached to the dialog node. Example: {'mousedown': someFunction}. There are a few special/simulated events of which you may also attach: 'focused', 'closed', 'opened', 'resized', and 'moved'. Except for 'close', these special event functions will be passed the dialog node as the only argument.

resizable If set to false the dialog will not be resizable (all dialogs are resizable by default). Note that if a dialog may not be resized it will also not be maximizable.

maximizable If set to false the dialog will not have a maximize icon.

minimizable If set to false the dialog will not have a minimize icon.

maximize Open the dialog maximized.

above If set to true the dialog will be kept above others.

data (object) If given, any contained properties will be set as 'data-*' attributes on the dialogContainer.

where If given, the dialog will be placed here (DOM node or querySelector-like string) and will only be able to move within the parent element. Otherwise the dialog will be appended to the Gate One container (GateOne.node) and will be movable anywhere on the page.

noEsc If true the dialog will not watch for the ESC key to close itself.

noTransitions If true CSS3 transitions will not be enabled for this dialog.

class Any additional CSS classes you wish to add to the dialog (space-separated).

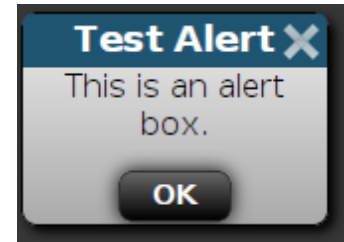
style Any CSS you wish to apply to the dialog. Example: {'style': {'width': '50%', 'height': '25%'}}

GateOne.Visual.**alert**(*title*, *message*[, *callback*])

Arguments

- **title** (*string*) – Title of the dialog that will be displayed.

- **message** – An HTML-formatted string or a DOM node; Main content of the alert dialog.
- **callback** (*function*) – A function that will be called after the user clicks “OK”.



Displays a dialog using the given *title* containing the given *message* along with an OK button. When the OK button is clicked, *callback* will be called.

```
>>> GateOne.Visual.alert('Test Alert', 'This is an alert box.');
```

Note: This function is meant to be a less-intrusive form of JavaScript’s `alert()`.

`GateOne.Visual.showDock(name)`

Opens up the given *name* for the user to view. If the dock does not already exist it will be created and added to the toolbar.

`GateOne.Visual.toggleOverlay()`

Toggles the overlay that visually indicates whether or not Gate One is ready for input. Normally this function gets called automatically by `GateOne.Input.capture()` and `GateOne.Input.disableCapture()` which are attached to `mousedown` and `blur` events, respectively.

`GateOne.Visual.enableOverlay()`

Displays an overlay above Gate One on the page that ‘greys it out’ to indicate it does not have focus. If the overlay is already present it will be left as-is.

The state of the overlay is tracked via the `GateOne.Visual.overlay` variable.

`GateOne.Visual.disableOverlay()`

Removes the overlay above Gate One (if present).

The state of the overlay is tracked via the `GateOne.Visual.overlay` variable.

GateOne.Storage

An object for opening and manipulating [IndexedDB](#) databases with fallback to [localStorage](#).

`GateOne.Storage.dbObject`

Arguments

- **DB** (*string*) – A string representing the name of the database you want to open.

Returns a new object that can be used to store and retrieve data stored in the given database. Normally you’ll get access to this object through the `GateOne.Storage.openDB()` function (it gets passed as the argument to your callback).

`GateOne.Storage.dbObject.get(storeName, key, callback)`

Retrieves the object matching the given *key* in the given object store (*storeName*) and calls *callback* with the result.

`GateOne.Storage.dbObject.put(storeName, value[, callback])`

Adds *value* to the given object store (*storeName*). If given, calls *callback* with *value* as the only argument.

`GateOne.Storage.dbObject.del(storeName, key[, callback])`

Deletes the object matching *key* from the given object store (*storeName*). If given, calls *callback* when the transaction is complete.

`GateOne.Storage.dbObject.dump(storeName, callback)`

Retrieves all objects in the given object store (*storeName*) and calls *callback* with the result.

`GateOne.Storage.init()`

Doesn't do anything (most init stuff for this module needs to happen before everything else loads).

`GateOne.Storage.cacheReady()`

Called when the fileCache DB has completed opening/initialization. Just sets `GateOne.Storage.fileCacheReady` to true.

`GateOne.Storage.cacheJS(fileObj)`

Stores the given *fileObj* in the 'fileCache' database in the 'js' store.

Note: Normally this only gets run from `GateOne.Utils.loadJSAction()`.

`GateOne.Storage.uncacheJS(fileObj)`

Removes the given *fileObj* from the cache (if present).

Note: This will fail silently if the given *fileObj* does not exist in the cache.

`GateOne.Storage.cacheStyle(fileObj, kind)`

Stores the given *fileObj* in the 'fileCache' database in the store associated with the given *kind* of stylesheet. Stylesheets are divided into different 'kind' categories because some need special handling (e.g. themes need to be hot-swappable).

Note: Normally this only gets run from `GateOne.Utils.loadStyleAction()`.

`GateOne.Storage.uncacheStyle(fileObj, kind)`

Removes the given *fileObj* from the cache matching *kind* (if present). The *kind* argument must be one of 'css', 'theme', or 'print'.

Note: This will fail silently if the given *fileObj* does not exist in the cache.

`GateOne.Storage.cacheExpiredAction(message)`

Attached to the `go:cache_expired` WebSocket action; given a list of *message['filenames']*, removes them from the file cache.

`GateOne.Storage.fileCheckAction(message)`

This gets attached to the `go:file_sync` WebSocket action; given a list of file objects which includes their modification times (*message['files']*) it will either load the file from the 'file-Cache' database or request the file be delivered via the (server-side) 'go:file_request' WebSocket action.

Note: Expects the 'fileCache' database be open and ready (normally it gets opened/initialized in `GateOne.initialize()`).

`GateOne.Storage.onerror(e)`

Attached as the errorback function in various storage operations; logs the given error (*e*).

`GateOne.Storage._upgradeDB(trans[, callback])`

DB version upgrade function attached to the `onupgradeneeded` event. It creates our object store(s).

If *callback* is given it will be called when the transaction is complete.

`GateOne.Storage.openDB(DB[, callback[, model[, version]]])`

Opens the given database (*DB*) for use and stores a reference to it as `GateOne.Storage.databases[DB]`.

If *callback* is given, will execute it after the database has been opened successfully.

If this is the first time we're opening this database a *model* must be given. Also, if the database already exists, the *model* argument will be ignored so it is safe to pass it with every call to this function.

If provided, the *version* of the database will be set. Otherwise it will be set to 1.

Example usage:

```
var model = {'BookmarksDB': {'bookmarks': {keyPath: "url"}, 'tags': {keyPath: "name"}}};
GateOne.Storage.openDB('somedb', function(dbObj) {console.log(dbObj)}), model);
// Note that after this DB is opened the IDBDatabase object will be available via GateOne.Storage.database
```

`GateOne.Storage.clearDatabase(DB[, storeName])`

Clears the contents of the given *storeName* in the given database (*DB*). AKA "the nuclear option."

If a *storeName* is not given the whole database will be deleted.

GateOne.User

The User module is for things like logging out, synchronizing preferences with the server, and it is also meant to provide hooks for plugins to tie into so that actions can be taken when user-specific events occur.

The following WebSocket actions are attached to functions provided by `GateOne.User`:

Action	Function
<code>go:gateone_user</code>	<code>GateOne.User.storeSessionAction()</code>
<code>go:set_username</code>	<code>GateOne.User.setUsernameAction()</code>
<code>go:applications</code>	<code>GateOne.User.applicationsAction()</code>
<code>go:user_list</code>	<code>GateOne.User.userListAction()</code>

`GateOne.User.init()`

Adds the user's ID (aka UPN) to the prefs panel along with a logout link.

`GateOne.User.workspaceApp(workspace)`

Attached to the `'go:switch_workspace'` event; sets `GateOne.User.activeApplication` to whatever application is attached to the `data-application` attribute on the provided *workspace*.

`GateOne.User.setActiveApp(app)`

Sets `GateOne.User.activeApplication` the given *app*.

Note: The *app* argument is case-insensitive. For example, if you pass `'terminal'` it will set the active application to `'Terminal'` (which is the name inside `GateOne.User.applications`).

`GateOne.User.setUsernameAction(username)`

Sets `GateOne.User.username` to *username*. Also triggers the `go:user_login` event with the username as the only argument.

Tip: If you want to call a function after the user has successfully loaded Gate One and authenticated attach it to the `go:user_login` event.

`GateOne.User.logout(redirectURL)`

This function will log the user out by deleting all Gate One cookies and forcing them to re-authenticate. By default this is what is attached to the 'logout' link in the preferences panel.

If provided, *redirectURL* will be used to automatically redirect the user to the given URL after they are logged out (as opposed to just reloading the main Gate One page).

Triggers the `go:user_logout` event with the username as the only argument.

`GateOne.User.storeSessionAction(message)`

This gets attached to the `go:gateone_user WebSocket` action in `GateOne.Net.actions`. It stores the incoming (encrypted) 'gateone_user' session data in `localStorage` in a nearly identical fashion to how it gets stored in the 'gateone_user' cookie.

Note: The reason for storing data in `localStorage` instead of in the cookie is so that applications embedding Gate One can remain authenticated to the user without having to deal with the cross-origin limitations of cookies.

`GateOne.User.applicationsAction()`

Sets `GateOne.User.applications` to the given list of *apps* (which is the list of applications the user is allowed to run).

`GateOne.User.preference(title, content)`

Adds a new section to the preferences panel using the given *title* and *content*. The *title* will be used to create a link that will bring up *content*. The *content* will be placed inside the preferences form.

To place a preference under a subsection (e.g. Terminal -> SSH) provide the *title* like so: "Terminal:SSH".

If *callback* is given it will be attached to the "go:save_prefs" event.

`GateOne.User.listUsers([callback])`

Sends the `terminal:list_users WebSocket` action to the server which will reply with the `go:user_list WebSocket` action containing a list of all users that are currently connected. Only users which are allowed to list users via the "list_users" policy will be able to perform this action.

If a *callback* is given it will be called with the list of users (once it arrives from the server).

`GateOne.User.userListAction()`

Attached to the `go:user_list WebSocket` action; sets `GateOne.User.userList` and triggers the `go:user_list` event passing the list of users as the only argument.

`GateOne.Events`

An object for event-specific stuff. Inspired by Backbone.js Events.

`GateOne.Events.on(events, callback[, context[, times]])`

Adds the given *callback* / *context* combination to the given *events*; to be called when the given *events* are triggered.

Arguments

- **events** (*string*) – A space-separated list of events that will have the given *callback* / *context* attached.
- **callback** (*function*) – The function to be called when the given *event* is triggered.

- **context** (*object*) – An object that will be bound to *callback* as this when it is called.
- **times** (*integer*) – The number of times this callback will be called before it is removed from the given *event*.

Examples:

```
>>> // A little test function
>>> var testFunc = function(args) { console.log('args: ' + args + ', this.foo: ' + this.foo) };
>>> // Call testFunc whenever the "test_event" event is triggered
>>> GateOne.Events.on("test_event", testFunc);
>>> // Fire the test_event with 'an argument' as the only argument
>>> GateOne.Events.trigger("test_event", 'an argument');
args: an argument, this.foo: undefined
>>> // Remove the event so we can change it
>>> GateOne.Events.off("test_event", testFunc);
>>> // Now let's pass in a context object
>>> GateOne.Events.on("test_event", testFunc, {'foo': 'bar'});
>>> // Now fire it just like before
>>> GateOne.Events.trigger("test_event", 'an argument');
args: an argument, this.foo: bar
```

`GateOne.Events.off(events, callback[, context])`

Removes the given *callback* / *context* combination from the given *events*

Arguments

- **events** (*string*) – A space-separated list of events.
- **callback** (*function*) – The function that's attached to the given events to be removed.
- **context** (*object*) – The context attached to the given event/callback to be removed.

Example:

```
>>> GateOne.Events.off("new_terminal", someFunction);
```

`GateOne.Events.once(events, callback[, context])`

A shortcut that performs the equivalent of `GateOne.Events.on(events, callback, context, 1)`.

`GateOne.Events.trigger(events)`

Triggers the given *events*. Any additional provided arguments will be passed to the callbacks attached to the given events.

Arguments

- **events** (*string*) – A space-separated list of events to trigger

Example:

```
>>> // The '1' below will be passed to each callback as the only argument
>>> GateOne.Events.trigger("new_terminal", 1);
```

`GateOne.restoreDefaults()`

Restores all of Gate One's user-specific prefs to default values. Primarily used in debugging Gate One.

`GateOne.openApplication(app[, settings[, where]])`

Opens the given *app* using its `__new__()` method.

If *where* is provided the application will be placed there. Otherwise a new workspace will be created and the app placed inside.

App The name of the application to open.

Settings Optional settings which will be passed to the application's `__new__()` method.

Where A `querySelector`-like string or DOM node where you wish to place the application.

`GateOne.Visual.disableTransitions(elem[, elem2[, ...]])`

Sets the 'noanimate' class on *elem* and any additional elements passed as arguments which can be a node or `querySelector`-like string (e.g. `#someid`). This class sets all CSS3 transformations to happen instantly without delay (which would animate).

`GateOne.Visual.enableTransitions(elem[, elem2[, ...]])`

Removes the 'noanimate' class from *elem* and any additional elements passed as arguments (if set) which can be a node or `querySelector`-like string (e.g. `#someid`).

Note: `GateOne.Input` was moved to a separate file (`gateone_input.js`) to reduce the size of `gateone.js` (since the input functions don't need to be available on the page right away).

`GateOne.Input`

`GateOne.Input` is in charge of all keyboard input as well as copy & paste stuff and touch events.

`GateOne.Input.init()`

Attaches our global `keydown`/`keyup` events and touch events

`GateOne.Input.modifiers(e)`

Given an event object, returns an object representing the state of all modifier keys that were held during the event:

```
{
  altgr: boolean,
  shift: boolean,
  alt:    boolean,
  ctrl:   boolean,
  meta:   boolean
}
```

`GateOne.Input.key(e)`

Given an event object, returns an object:

```
{
  type: e.type, // Just preserves it
  code: key_code, // Tries event.code before falling back to event.keyCode
  string: 'KEY_<key string>'
}
```

`GateOne.Input.kmouseey(e)`

Given an event object, returns an object:

```
{
  type: e.type, // Just preserves it
  left: boolean,
  right: boolean,
  middle: boolean,
}
```

`GateOne.Input.onKeyUp(e)`

Used in conjunction with `GateOne.Input.modifiers()` and `GateOne.Input.onKeyDown()` to emulate the meta key modifier using `KEY_WINDOWS_LEFT` and `KEY_WINDOWS_RIGHT` since "meta" doesn't work as an actual modifier on some browsers/platforms.

GateOne.Input.**onKeyDown**(*e*)

Handles keystroke events by determining which kind of event occurred and how/whether it should be sent to the server as specific characters or escape sequences.

Triggers the `go:keydown` event with keystroke appended to the end of the event (in lower case).

GateOne.Input.**onGlobalKeyUp**(*e*)

This gets attached to the 'keyup' event on `document.body`. Triggers the `global:keyup` event with keystroke appended to the end of the event (in lower case).

GateOne.Input.**onGlobalKeyDown**(*e*)

Handles global keystroke events (i.e. those attached to the window object).

GateOne.Input.**execKeystroke**(*e*, *global*)

Executes the keystroke or shortcut associated with the given keydown event (*e*). If *global* is true, will only execute global shortcuts (no regular keystroke overrides).

GateOne.Input.**registerShortcut**(*keyString*, *shortcutObj*)

Arguments

- **keyString** (*string*) – The `KEY_<key>` that will invoke this shortcut.
- **shortcutObj** (*object*) – A JavaScript object containing two properties: 'modifiers' and 'action'. See above for their format.

shortcutObj

param action A string to be `eval()`'d or a function to be executed when the provided key combination is pressed.

param modifiers An object containing the modifier keys that must be pressed for the shortcut to be called. Example: `{"ctrl": true, "alt": true, "meta": false, "shift": false}`.

Registers the given *shortcutObj* for the given *keyString* by adding a new object to `GateOne.Input.shortcuts`. Here's an example:

```
GateOne.Input.registerShortcut('KEY_ARROW_LEFT', {
  'modifiers': {
    'ctrl': true,
    'alt': false,
    'altgr': false,
    'meta': false,
    'shift': true
  },
  'action': 'GateOne.Visual.slideLeft()' // Can be an eval() string or a function
});
```

You don't have to provide *all* modifiers when registering a shortcut. The following would be equivalent to the above:

```
GateOne.Input.registerShortcut('KEY_ARROW_LEFT', {
  'modifiers': {
    'ctrl': true,
    'shift': true
  },
  'action': GateOne.Visual.slideLeft // Also demonstrating that you can pass a function instead of a string
});
```

Shortcuts registered via this function will only be usable when Gate One is active on the web page in which it is embedded. For shortcuts that need to *always* be usable see

```
GateOne.Input.registerGlobalShortcut().
```

Optionally, you may also specify a condition or Array of conditions to be met for the shortcut to be executed. For example:

```
GateOne.Input.registerShortcut('KEY_ARROW_LEFT', {
  'modifiers': {
    'ctrl': true,
    'shift': true
  },
  'conditions': [myCheckFunction, 'GateOne.Terminal.MyPlugin.isAlive'],
  'action': GateOne.Visual.slideLeft
});
```

In the example above the `GateOne.Visual.slideLeft` function would only be executed if `myCheckFunction()` returned `true` and if `'GateOne.Terminal.MyPlugin.isAlive'` existed and also evaluated to `true`.

`GateOne.Input.unregisterShortcut(keyString, shortcutObj)`

Removes the shortcut associated with the given *keyString* and *shortcutObj*.

`GateOne.Input.registerGlobalShortcut(keyString, shortcutObj)`

Used to register a *global* shortcut. Identical to `GateOne.Input.registerShortcut()` with the exception that shortcuts registered via this function will work even if `GateOne.prefs.goDiv` (e.g. `#gateone`) doesn't currently have focus.

Note: This function only matters when Gate One is embedded into another application.

`GateOne.Input.unregisterGlobalShortcut(keyString, shortcutObj)`

Removes the shortcut associated with the given *keyString* and *shortcutObj*.

`GateOne.Input.humanReadableShortcut(name, modifiers)`

Given a key *name* such as `'KEY_DELETE'` (or just `'G'`) and a *modifiers* object, returns a human-readable string. Example:

```
>>> GateOne.Input.humanReadableShortcut('KEY_DELETE', {"ctrl": true, "alt": true, "meta": false, "shift": false})
Ctrl-Alt-Delete
```

`GateOne.Input.humanReadableShortcutList(shortcuts)`

Given a list of *shortcuts* (e.g. `GateOne.Input.shortcuts`), returns an Array of keyboard shortcuts suitable for inclusion in a table. Example:

```
>>> GateOne.Input.humanReadableShortcutList(GateOne.Input.shortcuts);
[['Ctrl-Alt-G', 'Grid View'], ['Ctrl-Alt-N', 'New Workspace']]
```

`term_ww.js`

Todo

This!

1.3.3 Plugin Code

Gate One comes bundled with a number of plugins which can include any number of files in Python, JavaScript, or CSS (yes, you could have a CSS-only plugin!). These included plugins are below:

The Help Plugin

JavaScript

GateOne.Help

A global Gate One plugin for providing helpful/useful information to the user.

GateOne.Help.**aboutGateOne()**

Displays the Gate One version/credits.

GateOne.Help.**addHelpSection**(*section*, *title*, *action*[, *callback*])

Adds help to the Help panel under the given *section* using the given *title*. The *title* will be a link that performs one of the following:

- If *action* is a URL, showHelpSection() will be called to load the content at that URL.
- If *action* is a DOM node it will be displayed to the user (in the help panel).

Example:

```
>>> GateOne.Help.addHelp('Terminal', 'SSH Plugin', '/terminal/ssh/static/help.html');
```

If a *callback* is provided it will be called after the *action* is loaded (i.e. when the user clicks on the link).

GateOne.Help.**init()**

Creates the help panel and registers the Shift-F1 (show help) and Ctrl-S (displays helpful message about suspended terminal output) keyboard shortcuts.

GateOne.Help.**showFirstTimeDialog()**

Pops up a dialog for first-time users that shows them the basics of Gate One.

GateOne.Help.**showHelp()**

Displays the help panel.

GateOne.Help.**showHelpSection**(*helpURL*[, *callback*])

Shows the given help information (*helpURL*) by sliding out whatever is in the help panel and sliding in the new help text.

If *callback* is given it will be called after the content is loaded.

Note: The Terminal application has its own plugins.

1.3.4 Developing Plugins

Developing plugins for Gate One is easy and fun. See *The Example Plugin* for how it's done.

1.3.5 Embedding Gate One Into Other Applications

Embedding Gate One Into Other Applications

Gate One can be embedded into *any* web page (really!) *without* using an iframe. Since that's not a common thing to do on the web (yet) you'll need to understand a few things like how it works, what you can do with it, and how to keep it secure. This tutorial will teach you everything you need to know to get Gate One embedded into your web application.

To use the examples in this tutorial you must have Gate One installed on some host (*any* host) where you can run the 'gateone' command (you don't need root). Alternatively, you can run Gate One out of a directory (e.g. the GateOne directory from 'git clone') using the 'run_gateone.py' command.

How To Embed Gate One - Chapter 1

This part of the tutorial requires that you start your Gate One server using the following settings:

```
{
  "*": {
    "gateone": {
      // These are what's important for the tutorial:
      "origins": ["*"], // Disable origin checks (insecure but OK for a tutorial)
      "port": 8000, // The examples all use this port
      "url_prefix": "/",
      "auth": "none" // Note: This can be overridden by 20authentication.conf if you put it in 10server.conf
      // These settings are just to avoid conflicts with a regular Gate One installation:
      "cache_dir": "/tmp/gateone_tutorial_cache",
      "user_dir": "/var/lib/gateone/users",
      "session_dir": "/tmp/gateone_tutorial",
      "pid_file": "/tmp/gateone_tutorial.pid"
    }
  }
}
```

For convenience a 99tutorial_chapter1.conf file has already been created with these settings. Just copy it into a temporary *settings_dir* before starting Gate One:

```
# Assuming you downloaded Gate One to /tmp/GateOne...
user@host:/tmp/GateOne $ mkdir /tmp/chapter1 && cp gateone/docs/embedding_configs/99tutorial_chapter1.conf /tmp/chapter1/
user@host:/tmp/GateOne $ ./run_gateone.py --settings_dir=/tmp/chapter1
```

Before we continue please test your Gate One server by loading it in your browser. This will also ensure that you've accepted the server's SSL certificate (if necessary).

Warning: Gate One's SSL certificate must be trusted by clients in order to embed Gate One. In production you can configure Gate One to use the same SSL certificate as the website that has it embedded to avoid that problem. Just note, for that to work Gate One must be running at the same domain as the website that's embedding it. So if your website is <https://myapp.company.com/> your Gate One server would need to be running on a different port at myapp.company.com (e.g. <https://myapp.company.com:8000/>).

Placement Gate One needs to be placed inside an element on the page in order to work properly. This element will be where Gate One places <script> tags, preference panels, the toolbar (if enabled), and similar. Typically all you need is a div:

```
<div id="gateone"></div>
```

By default Gate One will assume you're placing all applications inside this element (aka 'the goDiv' or GateOne.node) so it will set it's style in such a way as to fill up the entirety of it's parent element. The idea is to make room for things like workspaces and terminals. For this part of the tutorial we'll place Gate One inside a div that has a fixed width and height and let it fill up that space:

```
<div id="gateone_container" style="width: 60em; height: 30em;">
  <div id="gateone"></div>
</div>
```

Note: You don't have to place terminals (or other Gate One applications) inside the #gateone container. More information about that is covered later in this tutorial.

Include gateone.js Before you can initialize Gate One on your web page you'll need to include gateone.js. You *could* just copy it out of Gate One's 'static' directory and include it in a <script> tag but it's usually a better idea to let Gate One serve up it's own gateone.js. This ensures that when you upgrade Gate One clients will automatically get the new file (less work).

```
<script src="https://your-gateone-server/static/gateone.js"></script>
```

Tip: You can also load the script on-demand via JS (if you know how). It doesn't use the window.onload event or similar.

Call GateOne.init() The GateOne.init() function takes some (optional) arguments but for this example all we need is url.

```
GateOne.init({url: "https://your-gateone-server/"});
```

Put that somewhere in your window.onload function and Gate One will automatically connect to the server, synchronize-and-load it's JavaScript/CSS, and open the New Workspace Workspace (aka the application selection screen).

Complete Example Here's an example of everything described above:

```
<!-- Include gateone.js somewhere on your page -->
<script src="https://gateone.mycompany.com/static/gateone.js"></script>

<!-- Decide where you want to put Gate One -->
<div id="gateone_container" style="position: relative; width: 60em; height: 30em;">
  <div id="gateone"></div>
</div>

<!-- Call GateOne.init() at some point after the page is done loading -->
<script>
window.onload = function() {
  // Initialize Gate One:
  GateOne.init({url: 'https://gateone.mycompany.com/'});
}
</script>
<!-- That's it! -->
```

Try It

API Authentication

Gate One includes an authentication API that can be used when embedded into other applications. It allows the application embedding Gate One to pre-authenticate users so they won't have to re-authenticate

when their browser connects to the Gate One server. Here's how it works:

Enable API Authentication Set `auth = "api"` in your `server.conf`:

```
# grep "^auth" server.conf
auth = "api"
```

Generate an API Key/Secret

```
# ./gateone.py --new_api_key
[I 120905 14:00:07 gateone:2679] A new API key has been generated: NDEzMWEwYTdlZTAzNDkxMWIwMDI4YzJmZTk4YzI4OWJjM
[I 120905 14:00:07 gateone:2680] This key can now be used to embed Gate One into other applications.
```

Note: The secret is not output to the terminal to avoid it being captured in session logs.

API keys and secrets are stored in your `30api_keys.conf` like so:

```
{
  "*": {
    "gateone": {
      "api_keys": {
        "<API Key>": "<Secret>",
        "<API Key 2>": "<Secret 2>"
      }
    }
  }
}
```

You'll need to have a look at your `30api_keys.conf` to see what the 'secret' is:

```
# cat settings/30api_keys.conf
{
  "*": {
    "gateone": {
      "api_keys": {
        "NDEzMWEwYTdlZTAzNDkxMWIwMDI4YzJmZTk4YzI4OWJjM": "M2U5YTMxMGQ3OWNlNDJlMTg5NmY0NmUyOTk5MmYwYWFiN"
      }
    }
  }
}
```

In the above example our API key would be, "NDEzMWEwYTdlZTAzNDkxMWIwMDI4YzJmZTk4YzI4OWJjM" and our API secret would be, "M2U5YTMxMGQ3OWNlNDJlMTg5NmY0NmUyOTk5MmYwYWFiN".

Tip: You can set the API Key and secret to whatever you like by editing your `30api_keys.conf`. By default they're random, 45-character strings but they can be any combination of characters other than colons and commas—even [Unicode](#)!. The following is a perfectly valid API key and secret:

```
"☐•☐•☐ ☐☐☐☐ ☐☐ ☐☐☐☐": "☐ ☐ ☐ ☐ ☐ → ☐ → ☐ ☐ ☐☐☐"
```

Generate An Auth Object The next step is to generate a [JSON](#) object (auth) from your application and pass it to `GateOne.init()`. The 'auth' object must contain the following information:

api_key The key that was generated when you ran `./gateone.py --new_api_key`

upn The username or userPrincipalName (aka UPN) of the user you wish to preauthenticate.

timestamp A JavaScript-style timestamp: 13 digits representing the amount of milliseconds since the epoch (January 1, 1970)

signature A valid [HMAC](#) signature that is generated from the `api_key`, `upn`, and `timestamp` (in that order).

signature_method The HMAC signature method that was used to sign the authentication object. Currently, only HMAC-SHA1 is supported.

api_version The version of Gate One's API authentication to use. Currently, only '1.0' is valid.

Here's an example 'auth' object:

```
authobj = {
  'api_key': 'MjkwYzc3MDI2MjhhNGZkNDg1MjJkODgyYjBmN2MyMTM4M',
  'upn': 'joe@company.com',
  'timestamp': '1323391717238',
  'signature': 'f6c6c82281f8d56797599aeee01a5e3efab05a63',
  'signature_method': 'HMAC-SHA1',
  'api_version': '1.0'
}
```

This object would then be passed to `GateOne.init()` like so:

```
GateOne.init({auth: authobj})
```

Assuming the signature is valid Gate One would then inherently trust that the user connecting over the WebSocket is [joe@company.com](#).

Note: Authentication objects (aka "authentication tokens") are only valid within the time frame specified in the `--api_timestamp_window` setting. They also can't be used more than once (to negate replay attacks).

Example API Authentication Code

The following are examples demonstrating how to generate valid 'auth' objects in various programming languages.

Python

```
import time, hmac, hashlib, json
secret = "secret"
authobj = {
  'api_key': "MjkwYzc3MDI2MjhhNGZkNDg1MjJkODgyYjBmN2MyMTM4M",
  'upn': "joe@company.com",
  'timestamp': str(int(time.time() * 1000)),
  'signature_method': 'HMAC-SHA1',
  'api_version': '1.0'
}
hash = hmac.new(secret, digestmod=hashlib.sha1)
hash.update(authobj['api_key'] + authobj['upn'] + authobj['timestamp'])
authobj['signature'] = hash.hexdigest()
valid_json_auth_object = json.dumps(authobj)
```

Here's a `create_signature()` function that can be used as a shortcut to those hash calls above:

```
def create_signature(secret, *parts):
    import hmac, hashlib
    hash = hmac.new(secret, digestmod=hashlib.sha1)
```

```
for part in parts:
    hash.update(str(part))
return hash.hexdigest()
```

...which could be used like so:

```
>>> create_signature(secret, api_key, upn, timestamp)
'f6c6c82281f8d56797599aeee01a5e3efab05a63'
```

PHP

```
$secret = 'secret';
$authobj = array(
    'api_key' => 'MjkwYzc3MDI2MjhhNGZkNDg1MjJkODgyYjBmN2MyMTM4M',
    'upn' => $_SERVER['REMOTE_USER'],
    'timestamp' => time() * 1000,
    'signature_method' => 'HMAC-SHA1',
    'api_version' => '1.0'
);
$authobj['signature'] = hash_hmac('sha1', $authobj['api_key'] . $authobj['upn'] . $authobj['timestamp'], $secret);
$valid_json_auth_object = json_encode($authobj);
```

Ruby

```
require 'cgi'
require 'openssl'
require 'json'
secret = 'secret'
authobj = {
    'api_key' => 'MjkwYzc3MDI2MjhhNGZkNDg1MjJkODgyYjBmN2MyMTM4M',
    'upn' => 'joe@company.com',
    'timestamp' => (Time.now.getutc.to_i * 1000).inspect,
    'signature_method' => 'HMAC-SHA1',
    'api_version' => '1.0'
}
authobj['signature'] = OpenSSL::HMAC.hexdigest('sha1', secret, authobj['api_key'] + authobj['upn'] + authobj['timestamp'])
valid_json_auth_object = JSON.generate(authobj)
```

1.4 Release Notes / Changelog

1.4.1 0.9

Release Date

October 13th, 2011

Summary of Changes

This was the initial release of Gate One. It is also known as “the beta”. There’s no comparison.

Note: Gate One 0.9 was written by Dan McDougall in his spare time over the course of ~9 months starting in January, 2011.

1.4.2 1.0

Release Date

March 6th, 2012

Summary of Changes

This was the first packaged release of Gate One with commercial support available from [Liftoff Software](#). Highlights of changes since the beta:

- **MAJOR performance enhancements:** Gate One is now much, much faster than the beta (server-side).
- **New Feature:** Added the ability to display PNG and JPEG images in terminals (e.g. `cat someimage.png`). Images output to stdout will be automatically resized to fit and displayed inline in a given terminal.
- **New Feature:** Added support for 256-color and aixterm (16-bit or 'bright') color modes.
- **New Feature:** Added support for PAM authentication.
- **New Feature:** Added IPv6 support.
- **New Feature:** Added the ability to duplicate SSH sessions without having to re-enter your password (OpenSSH Master mode with slaves aka session multiplexing).
- **New Feature:** Added the ability to redirect incoming HTTP connections on port 80 to HTTPS (whatever port is configured).
- **New Feature:** Added a command line log viewer (playback or flat view).
- The bookmarks manager is now feature-complete with sophisticated server synchronization. Bookmarks will stay in sync no matter what client you're connecting from.
- User sessions are now associated with the user account rather than the browser session. This means you can move from one desktop to another and go back to precisely what you were working on. You can even view both sessions simultaneously (live updates) if both browsers are logged in as the same user.
- Improved internationalization support. This includes support for translations and foreign language keyboards.
- Much improved terminal emulation. Display bugs with programs like midnight commander have been fixed.
- API-based authentication has been added to allow applications embedding Gate One to use their pre-authenticated users as-is without requiring the use of iframes or having to authenticate twice.
- Copy & Paste works in Windows now (it always worked fine in Linux).
- There are now dozens of hooks throughout the code for plugins to take advantage of. For example, you can now intercept the screen before it is displayed in the terminal to apply regular expressions and/or transformations to the text.
- A new "dark black" CSS theme has been added.
- Users can now select their CSS theme separate from their text color scheme.
- Added lots of helpful error messages in the event that a user is missing a dependency or doesn't have permission to, say, write to the user directory.

- Zillions of bugs have been fixed.

See the git commit log for full details on all changes.

1.4.3 1.1

Release Date

November 1st, 2012

Summary of Changes

- Now tracking changes in more detail.
- **MAJOR Performance Enhancement:** The terminal emulator now caches pre-rendered lines to provide significantly improved performance when rendering terminals into HTML (i.e. whenever there's a screen update). Example: In testing, the (localhost) round-trip time for keystroke→server→SSH→server→browser has been reduced from 70-150ms to 10-30ms.
- **MAC USERS REJOICE:** `⌘-c` and `⌘-v` now work to copy & paste!
- **Performance Enhancement:** The terminal emulator's data structures have been changed to significantly reduce memory utilization. Lists of lists of strings (`Terminal.screen`) and lists of lists of lists (`Terminal.renditions`) have been replaced with lists of unicode arrays (`array.array('u')` to be precise).
- **Performance Enhancement:** In the client JavaScript (*gateone.js*) terminal updates are handled much more efficiently: Instead of replacing the entire terminal `<pre>` with each update, individual lines are kept track of and updated independently of each other. This makes using full-screen interactive programs like vim much more responsive and natural. Especially with large browser window/terminal sizes.
- **Security Enhancement:** An `origins` setting has been added that will restrict which URLs client web browsers are allowed to connect to Gate One's WebSocket. This is to prevent an attacker from being able to control user's sessions via a (sophisticated) spear phishing attack.
- **Security Enhancement:** Logic has been added to prevent authentication replay attacks when Gate One is configured to use API authentication. Previous authentication signatures and timestamps will be checked before any provided credentials will be allowed.
- **Security Enhancement:** Gate One can now drop privileges to run as a different user and group. Continually running as root is no longer required—even if using a privileged port (`<=1024`).
- **Security Enhancement:** You can now require the use of client-side SSL certificates as an extra layer of security as part of the authentication process.
- **Embedding Enhancement:** Embedding Gate One into other applications is now much easier and there is an extensive tutorial available. To find out more see the `gateone/tests/hello_embedded` directory.
- **Plugin Enhancement:** Hooks have been added to allow plugins to modify Gate One's `index.html`. Arbitrary code can be added to the header and the body through simple variable declarations.
- **Plugin Enhancement:** Hooks have been added to allow plugins to modify the instances of `termio.Multiplex` and `terminal.Terminal` immediately after they are created (at runtime). This will allow plugin authors to, say, change how various file types are displayed or to add support for different kinds of terminal emulation.

- **New Plugin:** Mobile. This plugin allows Gate One to be used on mobile browsers that support WebSockets (Note: Only Mobile Firefox and Chrome for Android have been tested). Works best with devices that have a hardware keyboard.
- **New Plugin:** Example. It is heavily commented and provides examples of how to write your own Gate One plugin. Included are examples of how to use the new `widget()` function and how to track the deployment of your plugin. Try out the real-time load graph!
- **New Plugin:** Convenience. It contains a number of text transformations that make life in a terminal more convenient. Examples: Click on the bytes value in the output of `'ls -l'` and it will display a message indicating how much that value is in kilobytes, megabytes, gigabytes, terabytes, etc—whatever is appropriate (aka “human readable”). Click on a username in the output of `'ls -l'` and it will perform a lookup telling you all about that user. Ditto for group names; it even tells you which users have that group listed as their primary GID! It will also give you the ‘chmod equivalent’ of values like `'-rw-r--r--'` when that’s clicked as well. Lastly, it makes IPv4 and IPv6 addresses clickable: It will tell you what hostname they resolve to.
- **New Feature:** Added support for Python 3. NOTE: Gate One also runs on [pypy](#) and it’s very speedy!
- **New Feature:** Gate One now works in Internet Explorer! Well, IE 10 anyway.
- **New Feature:** Gate One can now detect and intelligently display PDF files just like it does for JPEG and PNG files.
- **New Feature:** Support for capturing the output of different kinds of files can now be added via a plugin. You can also override existing file types this way.
- **New Feature:** Gate One now includes init scripts for Debian/Ubuntu, Red Hat/CentOS, and Gentoo. These will be automatically installed via `setup.py`, the `deb`, or the `rpm`.
- **New Feature:** Gate One now keeps track of its own pid with the new `pid_file` option.
- **New Feature:** CSS/Styles are now downloaded over the WebSocket directly instead of merely being placed in the `<head>` of the current HTML page. This simplifies embedding.
- **New Feature:** Two new functions have been added to the SSH plugin that make it much easier to call and report on commands executed in a background session: `execRemoteCmd()` and `commandCompleted()`. See the documentation and the Example plugin for details.
- **New Feature:** A `widget()` function has been added to *gateone.js* that allows plugins to create elements that float above terminals. See the the documentation and the Example plugin (`example.js`) itself for details.
- **New Feature:** The bell sound is now downloaded over the WebSocket and cached locally in the user’s browser so it won’t need to be downloaded every time the user connects.
- **New Feature:** Users can now set a custom bell sound.
- **New Feature:** Most usage of the threading module in Gate One has been replaced with Tornado’s `PeriodicCallback` feature and multiprocessing (where appropriate). This is both more performant and reduces memory utilization considerably. Especially when there are a large number of open terminals.
- **New Feature:** Gate One can now be configured to listen on a Unix socket (as opposed to just TCP/IP addresses). Thanks to Tamer Mohammed Abdul-Radi of [Cloud9ers](#) for this contribution.
- **New Feature:** Old user session logs are now automatically removed after a configurable time period. See the `session_logs_max_age` option.
- **New Feature:** If you’ve set the number of rows/columns Gate One will now scale the size of each terminal in an attempt to fit it within the window. Looks much nicer than having a tiny-sized terminal in the upper left corner of the browser window.

- **New Feature:** Bookmarks can now be navigated via the keyboard. Ctrl-Alt-B will bring up the Bookmarks panel and you can then tab around to choose a bookmark.
- **New Feature:** Gate One now includes a print stylesheet so if you print out a terminal it will actually look nice and readable. This works wonderfully in conjunction with the “Printable” log view.
- **New Feature:** When copying text from a terminal it will now automatically be converted to plaintext (HTML formatting will be removed). It will also have trailing whitespace removed.
- **New Feature:** Added a new theme/text color scheme: Solarized. Thanks to Jakub Woyke for this contribution.
- **Themes:** Loads and loads of tweaks to improve Gate One’s overall appearance in varying situations.
- **Documentation:** Many pages of documentation have been added and its overall usefulness has been improved. For example, this changelog (□□□).

Notable Bugs Fixes

- **gateone.js:** You can now double-click to highlight a word in terminals in a very natural fashion. This is filed under bugs instead of new features because it was something that should’ve been working from the get-go but browsers are finicky beasts.
- **gateone.js:** No more crazy scrolling: The browser bug that would scroll text uncontrollably in terminals that had been moved down the page via a CSS3 transform has been worked around.
- **gateone.js:** Loads of bug fixes regarding embedding Gate One and the possibilities thereof. The hello_embedded tutorial is more than just a HOWTO; it’s a test case.
- **gateone.js:** The logic that detects the number of rows/columns that can fit within the browser window has been enhanced to be more accurate. This should fix the issue where the tops of terminals could get cut off under just the right circumstances.
- **gateone.js:** Fixed a bug where if you tried to drag a dialog in Firefox it would mysteriously get moved to the far left of the window (after which it would drag just fine). Now dialogs drag in a natural fashion.
- **gateone.js:** Fixed a bug where if you disabled terminal slide effects you couldn’t turn them back on.
- **gateone.js:** Fixed a bug with GateOne.Input.mouse() where it wasn’t detecting/assigning Firefox scroll wheel events properly.
- **gateone.js:** Fixed the bug with the - (hyphen-minus) key when using vim from inside ‘screen’. Note that this only seemed to happen on RHEL-based Linux distributions.
- **gateone.js:** Fixed the issue where you had to click twice on a terminal to move to it when in Grid view (only need to click once now).
- **gateone.js:** Fixed the bug where you could wind up with all sorts of HTML formatting when pasting in Mac OS X (and a few other paste methods). Pastes will now automatically be converted to plaintext if they’re registered by the browser as containing formatting.
- **gateone.py:** Terminal titles will now be set correctly when resuming a session.
- **gateone.py:** Generated self-signed SSL keys and certificates will now be stored in GATEONE_DIR instead of the current working directory unless absolute paths are provided via the –keyfile and –certificate options.
- **gateone.py:** When dtach=True and Gate One is stopped & started, resumed terminals will no longer be blank with incorrect values in \$ROWS and \$COLUMNS until you type ctrl=l. They should now appear properly and have the correct size set without having to do anything at all.

- **terminal.py:** Corrected the handling of unicode diacritics (accent marks that modify the preceding character) inside of escape sequences.
- **termio.py:** Fixed a bug where multi-regex patterns weren't working with `preprocess()`.
- **Logging Plugin:** The "View Log (Flat)" option (now renamed to "Printable Log") works reliably and looks nicer.
- **Playback Plugin:** Fixed the bug where a browser's memory utilization could slowly increase over time (only happened with Webkit-based browsers).
- **Playback plugin:** Fixed a bug where it was possible to get `UnicodeDecodeErrors` when exporting the current session's recording to HTML.
- **Playback Plugin:** Shift+scroll now works to go forwards/backwards in the playback history in Firefox. Previously this only worked in Chrome.
- **SSH Plugin:** Fixed a bug where the SSH Identity upload dialog wasn't working in Firefox (apparently Firefox uses 'name' instead of 'fileName' for file objects).
- **SSH Plugin:** In `ssh_connect.py`, fixed a bug with telnet connections where the port wasn't being properly converted to a string.

Other Notable Changes

- **EMBEDDED MODE CHANGES:** Embedded mode now requires manual invocation of many things that were previously automatic. For example, if you've set `embedded: true` when calling `GateOne.init()` you must now manually invoke `GateOne.Terminal.newTerminal()` at the appropriate time in your code (e.g. when a user clicks a button or when the page loads). See the `hello_embedded` tutorial for examples on how to use Embedded Mode.
- **gateone.py:** Added a new configuration option: `api_timestamp_window`. This setting controls how long to wait before an API authentication timestamp is no longer accepted. The default is 30 seconds.
- **gateone.py:** The dict that tracks things unique to individual browser sessions (i.e. where the 'refresh_timeout' gets stored) now gets cleaned up automatically when the user disconnects.
- **gateone.py:** You can now provide a *partial* `server.conf` before running Gate One for the first time (e.g. in packaging) and it will be used to set the provided values as defaults. After which it will overwrite your `server.conf` with the existing settings in addition to what was missing.
- **gateone.py:** If `dtach` support isn't enabled Gate One will now empty out the `session_dir` at exit.
- **gateone.py:** You may now designate which plugins are enabled by creating a `plugins.conf` file in `GATEONE_DIR`. The format of the file is, "one plugin name per line." Previously, to disable plugins you had to remove them from `GATEONE_DIR/plugins/`.
- **gateone.js:** From now on, when Gate One doesn't have focus (and isn't accepting keyboard input) a graphical overlay will "grey out" the terminal slightly indicating that it is no longer active. This should make it so that you always know when Gate One is ready to accept your keyboard input.
- **gateone.js:** From now on when you paste multiple lines into Gate One trailing whitespace will be removed from those lines. In 99% of cases this is what you want.
- **gateone.js:** Removed the Web Worker bug workaround specific to Firefox 8. Firefox has moved on.
- **gateone.js:** The timeout that calls `enableScrollback()` with each screen update has been modified to run after 500ms instead of 3500.
- **gateone.js:** Instead of emptying the scrollback buffer, `disableScrollback()` now just sets its style to "display: none;" and resets this when `enableScrollback()` is called.

- **gateone.js:** The “Info and Tools” and Preferences panels now have a close X icon in the upper right-hand corner like everything else.
- **gateone.js:** Added some capabilities checks so that people using inept browsers will at least be given a clear message as to what the problem is.
- **gateone.js:** From now on if you set the title of a terminal by hand it will not be overwritten by the `setTitleAction()` (aka the X11 title).
- **gateone.js:** The toolbar (icons) will now take the width of the scrollbar into account and be adjusted accordingly to make sure it isn’t too far to the left or overlapping the scrollbar.
- **gateone.js:** The toolbar will now scale in size proportionally to the `fontSize` setting. So if you are visually impaired and need a larger font size the toolbar icons will get bigger too to help you out.
- **gateone.js:** Added `GateOne.prefs.skipChecks` as an option that can be passed to `GateOne.init()`. If set to `true` it will skip all the capabilities checks/alerts that Gate One throws up if the browser doesn’t support something like WebSockets.
- **gateone.js:** You can now close panels and dialogs by pressing the ESC key.
- **gateone.js:** When Gate One is loaded from a different origin than where the server lives (i.e. when embedded) and the user has yet to accept the SSL certificate for said origin they will be presented with a dialog where they can accept it and continue. This should work around the problem of having to buy SSL certificates for all your Gate One servers.
- **gateone.js:** Added a `webWorker` option to `GateOne.prefs`. By default it will only be used when Gate One is unable to load the Web Worker via the Web Socket (i.e. via a `Blob()`). This usually only happens on older versions of Firefox and IE 10, specifically. Also, it will *actually* only need to be set if you’re embedding Gate One into another application that is listening on a different port than the Gate One server (I know, right?). It is a very, very specific situation in which it would be required.
- **gateone.js:** Lots of minor API additions and changes. Too many to list; you’ll just have to look at the docs. See: *gateone.js*.
- **gateone.js:** When the screen updates while viewing the scrollbar buffer it will no longer automatically scroll to the bottom of the view. If a keystroke is pressed *that* will scroll to the bottom. This should allow one to scroll up while something is outputting lines to the terminal without having the scrolling behavior interrupt what you’re looking at.
- **gateone.js:** When `<script>` tags are included in the incoming terminal screen they will be executed automatically. Very convenient for plugins that override HTML output of `FileType` magic in `terminal.py`.
- **go_process.js:** Before loading lines on the screen the Web Worker will now strip trailing whitespace. This should make copying & pasting easier.
- **index.html:** Changed `{{js_init}}` to be `{% raw js_init %}` so people don’t have to worry about Tornado’s template engine turning things like quotes into HTML entities.
- **logviewer.py:** The functions that play back and display `.golog` files have been modified to read log data in chunks to save huge amounts of memory. Playing back or displaying a gigantic log should now use as much memory as a small one (i.e. very little).
- **terminal.py:** Improved the ability of `Terminal.write()` to detect and capture images by switching from using `re.match()` to using `re.search()`.
- **terminal.py:** The mechanism that detects certain files being output to the terminal has been re-worked: It is now much easier to add support for new file types by subclassing the new `FileType` class and calling `Terminal.add_magic()`.

- **terminal.py:** Added a new global function: `css_colors()`. It just dumps the CSS style information for all the text colors that `Terminal.dump_html()` supports. The point is to make it easier for 3rd party apps to use `dump_html()`.
- **terminal.py:** Added a new global at the bottom of the file: `CSS_COLORS`. It holds all the CSS classes used by the new `css_colors()` function.
- **termio.py:** Lots of improvements to the way `.golog` files are generated. Logging to these files now requires less resources and happens with less CPU overhead.
- **termio.py:** Added the IUTF-8 setting (and similar) via `termios` when the “command” is forked/executed. This should ensure that multi-byte Unicode characters are kept track of properly in various erasure scenarios (e.g. backspace key, up/down arrow history, etc). Note that this doesn't work over SSH connections (it's an OpenSSH bug).
- **termio.py:** Instances of `Multiplex()` may now attach an `exitfunc` that does exactly what you'd expect: It gets called when the spawned program is terminated.
- **termio.py:** You can now pass a string as the 'callback' argument to `Multiplex.expect()` and it will automatically be converted into a function that writes said string to the child process.
- **termio.py:** Changed `Multiplex.writeline()` and `Multiplex.writelines()` so they write `\r\n` instead of just `\n`. This should fix an issue with terminal programs that expect keystrokes instead of just newlines.
- **termio.py:** The rate limiter will no longer truncate the output of terminal applications. Instead it simply suspends their output for ten seconds at a time. This suspension can be immediately interrupted by the user pressing a key (e.g. Ctrl-C).
- **termio.py:** The functions that handle how logs are finalized have been modified to reduce memory consumption by orders of magnitude. For example, when finalizing a humongous `.golog`, the `get_or_update_metadata()` function will now read the file in chunks and be very conservative about the whole process instead of reading the entire log into memory before performing operations.
- **utils.py:** Increased the timeout value on the `openssl` commands since the default 5-second timeout wasn't long enough on slower systems.
- **Playback Plugin:** The logic that adds the playback controls has been modified to use the new `GateOne.prefs.rowAdjust` property (JavaScript).
- **Playback plugin:** Whether or not the playback controls will appear can now be configured via the `GateOne.prefs.showPlaybackControls` option. So if you're embedding Gate One and don't want the playback controls just pass `showPlaybackControls: false` to `GateOne.init()`.
- **SSH Plugin:** In `ssh_connect.py`, added a check to make sure that the user's 'ssh' directory is created before it starts trying to use it.
- **SSH Plugin:** `execRemoteCmd()` now supports an `errorback` function as a fourth argument that will get called in the event that the remote command execution isn't successful.
- **SSH Plugin:** Generating the public key using the private key is now handled asynchronously (so it won't block on a slow or bogged-down system).
- **SSH Plugin:** Private keys will now be validated before they're saved. If a key does not pass (basic) validation an error will be presented to the user and nothing will be saved.
- **SSH Plugin:** The user will now be asked for the passphrase of the private key if they do not provide a public key when submitting the identity upload form. This is so the public key can be generated from the private one (and it sure beats a silent failure).
- **Help Plugin:** When viewing “About Gate One” it will now show which version you're running (based on the version string of the `GateOne` object in `gateone.js`).

See the git commit log for full details on all changes.

a

app_terminal, ??

b

bookmarks, ??

e

example, ??

g

gateone.auth.authentication, ??

gateone.auth.authorization, ??

gateone.auth.ctypes_pam, ??

gateone.auth.pam, ??

gateone.auth.sso, ??

gateone.core.log, ??

gateone.core.server, ??

gateone.core.utils, ??

l

logging_plugin, ??

logviewer, ??

n

notice, ??

p

playback, ??

s

ssh, ??

t

terminal, ??

termio, ??

w

woff_info, ??