Girder Documentation

Release 2.0.0

Kitware

October 10, 2016

Contents

CHAPTER 1

What is Girder?

Girder is a free and open source web-based **data management platform** developed by Kitware as part of the Resonant data and analytics ecosystem. What does that mean? Girder is both a standalone application and a platform for building new web services. It's meant to enable quick and easy construction of web applications that have some or all of the following requirements:

- **Data organization and dissemination** Many web applications need to manage data that are dynamically provided by users of the system, or exposed through external data services. Girder makes construction and organization of dynamic data hierarchies simple. One of the most powerful aspects of Girder is that it can transparently store, serve, and proxy data from heterogeneous backend storage engines through a single RESTful web API, including local filesystems, MongoDB databases, Amazon S3-compliant key-value stores, and Hadoop Distributed Filesystems (HDFS).
- **User management & authentication** Girder also includes everything needed for pluggable user management and authentication out of the box and adheres to best practices in web security. The system can be configured to securely store credentials itself, or defer to third-party authentication services such as OAuth or LDAP.
- **Authorization management** Girder supports a simple access control scheme that allows both user-based and role-based access control on resources managed in the system. The project has undergone rigorous security audits and has extensive automated testing to exercise authorization behavior and ensure correctness.

For an overview of the concepts present in Girder, we recommend checking out the User Guide.

Girder is published under the Apache 2.0 License. Its source code can be found at https://github.com/girder/girder.

CHAPTER 2

The architecture

Girder's server-side architecture is focused around the construction of RESTful web APIs to afford minimal coupling between the backend services and the frontend clients. This decoupling allows multiple clients all to use the same server-side interface. While Girder does contain its own single-page javascript web application, the system can be used by any HTTP-capable client, either inside or outside of the web browser environment. Girder can even be run without its front-end application present at all, only serving the web API routes.

The web API is mostly used to interact with resources that are represented by **models** in the system. Models internally interact with a Mongo database to store and retrieve persistent records. The models contain methods for creating, changing, retrieving, and deleting those records. The core Girder model types are described in the *Concepts* section of the user guide.

The primary method of customizing and extending Girder is via the development of **plugins**, the process of which is described in the Plugin Development section of this documentation. Plugins can, for example, add new REST routes, modify or remove existing ones, serve up a different web application from the server root, hook into model lifecycle events or specific API calls, override authentication behavior to support new authentication services or protocols, add a new backend storage engine for file storage, or even interact with a completely different DBMS to persist system records – the extent to which plugins are allowed to modify and extend the core system behavior is nearly limitless.

Plugins are self-contained in their own directory within the Girder source tree. Therefore they can reside in their own separate source repository, and are installed by simply copying the plugin source tree under an existing Girder installation's *plugins* directory. The Girder repository contains several generally useful plugins out of the box, which are described in the Plugins section.

CHAPTER 3

Table of contents

3.1 Administrator Documentation

3.1.1 System Prerequisites

The following software packages are required to be installed on your system:

- Python 2.7 or 3.4
- pip
- MongoDB 2.6+
- Node.js
- curl
- zlib
- libjpeg

Additionally, in order to send out emails to users, Girder will need to be able to communicate with an SMTP server. Proper installation and configuration of an SMTP server on your system is beyond the scope of these docs, but we recommend setting up Postfix.

See the specific instructions for your platform below.

Note: We perform continuous integration testing using Python 2.7 and Python 3.4. The system *should* work on other versions of Python 3 as well, but we do not verify that support in our automated testing at this time, so use at your own risk.

Warning: Some Girder plugins do not support Python 3 at this time due to third party library dependencies. Namely, the HDFS Assetstore plugin and the Metadata Extractor plugin will only be available in a Python 2.7 environment.

Debian / Ubuntu

Install the prerequisites using APT:

```
sudo apt-get install curl g++ git libffi-dev make python-dev python-pip libjpeg-dev zlib1g-dev
```

MongoDB 2.6 requires a special incantation to install at this time. Install the APT key with the following:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

For Debian, create the following configuration file for the MongoDB APT repository:

```
echo 'deb http://downloads-distro.mongodb.org/repo/debian-sysvinit dist 10gen' \
    | sudo tee /etc/apt/sources.list.d/mongodb.list
```

For Ubuntu, instead create the following configuration file:

```
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' \
    | sudo tee /etc/apt/sources.list.d/mongodb.list
```

Reload the package database and install MongoDB server using APT:

```
sudo apt-get update
sudo apt-get install mongodb-org-server
```

With Ubuntu 16.04, create a systemd init script at

`/lib/systemd/system/mongod.service`:

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
Documentation=https://docs.mongodb.org/manual

[Service]
User=mongodb
Group=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf

[Install]
WantedBy=multi-user.target
```

and start it with:

```
sudo service mongod start
```

Enable the Node.js APT repository:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
```

Install Node.js and NPM using APT:

```
sudo apt-get install nodejs
```

Note: It's recommended to get the latest version of the npm package manager, and Girder currently requires at least version 3 of npm. To upgrade to the latest npm, run:

```
npm install -g npm
```

This may need to be run as root using `sudo`.

CentOS / Fedora / Red Hat Enterprise Linux

For CentOS and Red Hat Enterprise Linux, enable the Extra Packages for Enterprise Linux YUM repository:

```
sudo yum install epel-release
```

Install the prerequisites using YUM:

```
sudo yum install curl gcc-c++ git libffi-devel make python-devel python-pip libjpeg-turbo-devel zlib-
```

Create a file `/etc/yum.repos.d/mongodb.repo` that contains the following configuration information for the MongoDB YUM repository:

```
[mongodb]
name=MongoDB Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
gpgcheck=0
enabled=1
```

Install MongoDB server using YUM:

```
sudo yum install mongodb-org-server
```

Enable the Node.js YUM repository:

```
curl -sL https://rpm.nodesource.com/setup_4.x | sudo bash -
```

Install Node.js and NPM using YUM:

```
sudo yum install nodejs
```

Arch Linux

For Arch Linux it is important to note that Python 3 is default. This means that most commands will need a 2 appending to them, i.e. python2, pip2, ...

Install the prerequisites using the pacman tool:

```
sudo pacman -S python2 python2-pip mongodb nodejs
```

OS X

It is recommended to use Homebrew to install the required packages on OS X.

To install all the prerequisites at once just use:

```
brew install python mongodb node
```

Note: OS X ships with Python in `/usr/bin`, so you might need to change your PATH or explicitly run `/usr/local/bin/python` when invoking the server so that you use the version with the correct site packages installed.

Windows

Warning: Windows is not supported or tested. This information is provided for developers. Use at your own risk.

Download, install, and configure MongoDB server following the instructions on the MongoDB website, and download and run the Node.js Windows Installer from the Node.js website.

Download and install the Windows MSI Installer for the latest Python 2 release from the Python website, and then download and run the ez_setup.py bootstrap script to install Setuptools for Python. You may need to add `python\scripts` to your path for NPM to work as expected.

From a command prompt, install pip:

```
easy_install pip
```

If bcrypt fails to install using pip (e.g., with Windows 7 x64 and Python 2.7), you need to manually install it prior to installing girder. You can build the package from source or download a wheel file from https://bitbucket.org/alexandrul/py-bcrypt/downloads and install it with the following:

```
pip install wheel
pip install py_bcrypt.whl
```

3.1.2 Installation

Before you install, see the System Prerequisites guide to make sure you have all required system packages installed.

Creating a virtual environment

While not strictly required, it is recommended to install Girder within its own virtual environment to isolate its dependencies from other python packages. To generate a new virtual environment, first install/update the `virtualenv` and `pip` packages

```
sudo pip install -U virtualenv pip
```

Now create a virtual environment using the virtualenv command. You can place the virtual environment directory wherever you want, but it should not be moved. The following command will generate a new directory called `girder_env` in your home directory:

```
virtualenv ~/girder_env
```

Now you can run `source ~/girder_env/bin/activate` to *enter* the new virtual environment. Inside the virtual environment you can use `pip`, `python`, and any other python script installed in your path as usual. You can exit the virtual environment by running the shell function `deactivate`. The shell variable `VIRTUAL_ENV` will also list the absolute path to the current virtual environment. Entering a virtual environment only persists for your current shell, so you must source the activation script again whenever you wish to enter within a new shell session. Users and developers needing to work on several virtual environments should consider using other packages that help manage them such as virtualenvwrapper, autoenv, pyenv-virtualenv, or pyenv-virtualenvwrapper.

Install with pip

To install the Girder distribution from the python package index, simply run

```
pip install girder
```

This will install the core Girder server as a site package in your system or virtual environment. At this point, you might want to check the configuration to change your plugin and logging paths. In order to use the web interface, you must also install the web client libraries. Girder installs a python script that will automatically build and install these libraries for you. Just run the following command:

```
girder-install web
```

Note: Installing the web client code requires the node package manager (npm). See the System Prerequisites section for instructions on installing nodejs.

Note: If you installed Girder into your system `site-packages`, you may need to run the above commands as root.

Once this is done, you are ready to start using Girder as described in this section: *Run*.

Installing extra dependencies with pip

Girder comes bundled with a number of Plugins that require extra python dependencies in order to use. By default, none of these dependencies will be installed; however, you can tell pip to install them using pip's "extras " syntax. Each girder plugin requiring extra python dependencies can be specified during the pip install. For example, installing girder with support for the *celery_jobs* and *geospatial* plugins can be done like this:

```
pip install girder[celery_jobs,geospatial]
```

There is also an extra you can use to install the depencies for all bundled plugins supported in the current python environment called `plugins`:

```
pip install girder[plugins]
```

Warning: Not all plugins are available in every python version and platform. Specifying a plugin for in an unsupported environment will raise an error.

Install from Git Checkout

Obtain the Girder source code by cloning the Git repository on GitHub:

```
git clone https://github.com/girder/girder.git
cd girder
```

To run the server, you must install some external Python package dependencies:

```
pip install -e .
```

or:

```
pip install -e .[plugins]
```

to install the plugins as well.

Note: This will install the most recent versions of all dependencies. You can also try to run `pip install -r requirements.txt` to duplicate the exact versions used by our CI testing environment; however, this can lead to problems if you are installing other libraries in the same virtual or system environment.

To build the client-side code project, cd into the root of the repository and run:

```
girder-install web
```

This will run multiple Grunt tasks, to build all of the Javascript and CSS files needed to run the web client application.

Run

To run the server, first make sure the Mongo daemon is running. To manually start it, run:

```
mongod &
```

Then to run Girder itself, just use the following command:

```
girder-server
```

Then open http://localhost:8080/ in your web browser, and you should see the application.

Initial Setup

The first user to be created in the system is automatically given admin permission over the instance, so the first thing you should do after starting your instance for the first time is to register a user. After that succeeds, you should see a link appear in the navigation bar that says `Admin console`.

The next recommended action is to enable any plugins you want to run on your server. Click the `Admin console` navigation link, then click `Plugins`. Here, you can turn plugins on or off. Whenever you change the set of plugins that are enabled, you must restart the CherryPy server for the change to take effect. For information about specific plugins, see the *Plugins* section.

After you have enabled any desired plugins and restarted the server, the next recommended action is to create an `Assetstore` for your system. No users can upload data to the system until an assetstore is created, since all files in Girder must reside within an assetstore. See the *Assetstores* section for a brief overview of `Assetstores`.

Installing third-party plugins

Girder ships with a *standard library of plugins* that can be enabled in the admin console, but it's common for Girder installations to require additional third-party plugins to be installed. If you're using a pip installed version of Girder, you can simply use the following command:

```
girder-install plugin /path/to/your/plugin
```

That command will expose the plugin to Girder and build any web client targets associated with the plugin. You will still need to enable it in the console and then restart the Girder server before it will be active.

Note: The `girder-install plugin` command can also accept a list of plugins to be installed. You may need to run it as root if you installed Girder at the system level.

For development purposes it is possible to symlink (rather than copy) the plugin directory. This is accomplished with the `-s` or `--symlink` flag:

```
girder-install -s plugin /path/to/your/plugin
```

Enabled plugins installed with `-s` may be edited in place and those changes will be reflected after a server restart.

3.1.3 Deploy

There are many ways to deploy Girder into production. Here is a set of guides on how to deploy Girder to several different platforms.

Heroku

This guide assumes you have a Heroku account and have installed the Heroku toolbelt.

Girder contains the requisite Procfile, buildpacks, and other configuration to be deployed on Heroku. To deploy Girder to your Heroku space, run the following commands. We recommend doing this on your own fork of Girder to keep any customization separate.

```
$ cd /path/to/girder/tree
$ heroku apps:create your_apps_name_here
$ heroku config:add BUILDPACK_URL=https://github.com/ddollar/heroku-buildpack-multi.git
$ heroku addons:add mongolab
$ git remote add heroku git@heroku.com:your_apps_name_here.git
$ git push heroku
$ heroku open
```

You should now see your Girder instance running on Heroku. Congratulations!

Reverse Proxy

In many cases, it is useful to route multiple web services from a single server. For example, if you have a server accepting requests at `www.example.com`, you may want to forward requests to `www.example.com/girder` to a Girder instance listening on port `9000`.

Apache

When using Apache, configure Apache's mod_proxy to route traffic to these services using a reverse proxy. Add the following section to your Apache config:

```
<VirtualHost *:80>
    ProxyPass /girder http://localhost:9000
    ProxyPassReverse /girder http://localhost:9000
</VirtualHost>
```

Nginx

Nginx can be used by adding a block such as:

```
location /girder/ {
    proxy_set_header Host $proxy_host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Proto $scheme;
```

```
    proxy_pass http://localhost:9000/;
    # Must set the following for SSE notifications to work
    proxy_buffering off;
    proxy_cache off;
    proxy_set_header Connection '';
    proxy_http_version 1.1;
    chunked_transfer_encoding off;
    proxy_read_timeout 600s;
    proxy_send_timeout 600s;
    # proxy_request_buffering option only works on nginx >= 1.7.11
    # but is necessary to support streaming requests
    proxy_request_buffering off;
}
```

And under the containing `server` block, make sure to add the following rule:

```
server {
    client_max_body_size 500M;
    ...
}
```

Girder Settings

In such a scenario, Girder must be configured properly in order to serve content correctly. This can be accomplished by setting a few parameters in your local configuration file at `girder/conf/girder.local.cfg`. In this example, we have the following:

```
[global]
server.socket_host: "0.0.0.0"
server.socket_port: 9000
tools.proxy.on: True

[server]
api_root: "/girder/api/v1"
static_root: "/girder/static"
```

Note: If your chosen proxy server does not add the appropriate `X-Forwarded-Host` header (containing the host used in http requests, including any non-default port to proxied requests), the `tools.proxy.base` and `tools.proxy.local` configuration options must also be set in the `[global]` section as:

```
tools.proxy.base: "http://www.example.com/girder"
tools.proxy.local: ""
```

After modifying the configuration, always remember to rebuild Girder by changing to the Girder directory and issuing the following command:

```
$ npm install && npm run build
```

Docker Container

Every time a new commit is pushed to master, Docker Hub is updated with a new image of a docker container running Girder. This container exposes Girder at port 8080 and requires the database URL to be passed in as an option. For more information, see the Docker Hub Page. Since the container does not run a database, you'll need to run a command in the form:

```
$ docker run -p 8080:8080 girder/girder -d mongodb://db-server-external-ip:27017/girder
```

Google Container Engine

Google Container Engine lets you host and manage Docker containers on Google Compute Engine instances. Before following the instructions here, follow Google's tutorial for setting up Wordpress, which will make the following steps more clear.

We will assume you have performed `gcloud auth login` and the following environment variables set:

```
$ export ZONE=us-central1-a
$ export CLUSTER_NAME=hello-girder
```

Start a new project in Google Developers Console (here we assume its identifier is `my-girder`). Set this as your active project with

```
$ gcloud config set project my-girder
```

Now click the Container Engine menu item on the left of the console to initialize the container service, then create a new cluster with:

```
$ gcloud preview container clusters create $CLUSTER_NAME --num-nodes 1 --machine-type n1-standard-2 -
```

This will create two instances, a master and a worker:

```
$ gcloud compute instances list --zone $ZONE
NAME                     ZONE          MACHINE_TYPE  INTERNAL_IP   EXTERNAL_IP    STATUS
k8s-hello-girder-master us-central1-a n1-standard-2 X.X.X.X       X.X.X.X        RUNNING
k8s-hello-girder-node-1 us-central1-a n1-standard-2 X.X.X.X       X.X.X.X        RUNNING
```

The worker will hold our Docker containers, MongoDB and Girder. The worker needs some extra storage than the standard 10GB, so let's make a new 100GB storage drive and attach it to our worker:

```
$ gcloud compute disks create mongodb --size 100GB --zone $ZONE
$ gcloud compute instances attach-disk k8s-hello-girder-node-1 --disk mongodb --zone $ZONE
```

Now we need to ssh into our worker node, which you can do from the Developers Console, and mount the disk to `/data`. First we find the name of the device, here `sdb`.

```
user_name@k8s-hello-girder-node-1:~$ ls -l /dev/disk/by-id/google-*
lrwxrwxrwx 1 root root  9 Nov 22 20:31 /dev/disk/by-id/google-mongodb -> ../../sdb
lrwxrwxrwx 1 root root  9 Nov 22 19:32 /dev/disk/by-id/google-persistent-disk-0 -> ../../sda
lrwxrwxrwx 1 root root 10 Nov 22 19:32 /dev/disk/by-id/google-persistent-disk-0-part1 -> ../../sda1
```

Then we create the directory and mount the drive:

```
user_name@k8s-hello-girder-node-1:~$ sudo mkdir /data
user_name@k8s-hello-girder-node-1:~$ sudo /usr/share/google/safe_format_and_mount -m "mkfs.ext4 -F" /
```

Now we are ready to install our pod, which is a collection of containers that work together. Save the following yaml specification for our MongoDB/Girder pod to `pod.yaml`:

```
---
    version: v1beta1
    id: girder
    kind: Pod
    desiredState:
        manifest:
            version: v1beta2
```

```
            containers:
              -
                name: mongodb
                image: dockerfile/mongodb
                ports:
                  -
                    name: db
                    containerPort: 27017
                volumeMounts:
                  -
                    name: data
                    mountPath: /data/db
              -
                name: application
                image: girder/girder
                ports:
                  -
                    name: app
                    containerPort: 8080
                    hostPort: 80
            volumes:
              -
                name: data
                source:
                    hostDir:
                        path: /data/db
```

Note that we are letting MongoDB use the host's `/data` directory, which will have more space and will persist even if our containers are shut down and restarted. Start the pod back at your local command line with:

```
$ gcloud preview container pods --cluster-name $CLUSTER_NAME create girder --zone $ZONE --config-file
```

You can check the status of your pod with:

```
$ gcloud preview container pods --cluster-name $CLUSTER_NAME describe girder --zone $ZONE
ID          Image(s)                          Host
----------  ----------                        ----------
girder      dockerfile/mongodb,girder/girder  k8s-hello-girder-node-1.c.hello-girder.internal/X.X.X.X
```

Add a firewall rule to expose port 80 on your worker:

```
$ gcloud compute firewall-rules create hello-girder-node-80 --allow tcp:80 --target-tags k8s-hello-gi
```

After everything starts, which may take a few minutes, you should be able to visit your Girder instance at `http://X.X.X.X` where `X.X.X.X` is the IP address in the container description above. Congratulations, you have a full Girder instance available on Google Container Engine!

3.1.4 Configuration

In many cases, Girder will work with default configuration whether installed via pip or from a source checkout or tarball. That said, the Girder config file can be set at the following locations (ordered by precedent):

1. The path specified by the environment variable *GIRDER_CONFIG*.
2. ~/.girder/girder.cfg
3. /etc/girder.cfg
4. /path/to/girder/package/conf/girder.local.cfg

5. /path/to/girder/package/conf/girder.dist.cfg

Logging

Much of Girder's output is placed into the error or info log file. By default, these logs are stored in ~/.girder/logs. To set the Girder log root or error and info logs specifically, set the *log_root*, *error_log_file*, and/or *info_log_file* variables in the *logging* config group. If *log_root* is set, error and info will be set to error.log and info.log within *log_root* respectively. The *_log_file* variables will override that setting and are *absolute* paths.

Server thread pool

Girder can handle multiple requests at one time. The maximum number of simultaneous requests is set with the *server.thread_pool* value in the *global* config group. Once this many connections have been made to Girder, additional connections will block until existing connections finish.

Most operations on Girder are quick, and therefore do not use up a connection for a long duration. Some connections, notably calls to the *notification/stream* endpoint, can block for long periods. If you expect to have many clients, either increase the size of the thread pool or switch to using intermittent polling rather than long-duration connections.

Each available thread uses up some additional memory and requires internal socket or handle resources. The exact amount of memory and resources is dependent on the host operating system and the types of queries made to Girder. As one benchmark from an Ubuntu server, each additional available but unused connection requires roughly 25 kb of memory. If all connections are serving notification streams, each uses around 50 kb of memory.

Changing file limits

If all server threads are in use, additional attempts to connect will use a file handle while waiting to be processed. The number of open files is limited by the operating system, and may need to be increased. This limit affects actual connections, pending connections, and file use.

The method of changing file limits varies depending on your operating system. If your operating system is not listed here, try a web search for "Open Files Limit" along with your OS's name.

Linux You can query the current maximum number of files with the command:

```
ulimit -Sn
```

To increase this number for all users, as root or with sudo privileges, edit `/etc/security/limits.conf` and append the following lines to the end of the file:

```
*    soft    nofile    32768
*    hard    nofile    32768
```

Save and close the file. The user running the Girder server will need to logout and log back in and restart the Girder server for the new limits to take effect.

This raises the limits for all users on the system. You can limit this change to just the user that runs the Girder server. See the documentation for `/etc/security/limits.conf` for details.

Managing Routes

When plugins which have their own custom webroot are enabled, they are mounted at /pluginName. In certain cases it may be desirable for the site administrator to mount such plugins at their own specified paths.

These paths can be modified by navigating to Admin Console -> Server Configuration and visiting the Routing section.

3.1.5 Provisioning

Girder is packaged for provisioning through the popular IT automation tool Ansible.

Specifically, Girder is available as an Ansible role to be fetched through Ansible Galaxy. This allows for a user to point their own Ansible playbook at a number of servers and deploy Girder with a single command. Provided in Girder are a number of example playbooks to demonstrate various ways Girder can be deployed using Ansible.

To test these roles, one can use Vagrant with any of our playbooks to deploy Girder to a live machine.

Note: Our playbooks are only currently supporting Ubuntu 14.04 and require Ansible >= 2.1.1.

Example: Deploying Girder with NGINX to a VirtualBox

Assuming a working copy of Girder is in your current directory:

```
GIRDER_EXAMPLE=girder-nginx vagrant up
```

Note: A full list of examples that can be used exist in devops/ansible/examples.

After a few minutes of running you should have a functioning Girder instance sitting behind an NGINX proxy at http://localhost:9080.

Using Ansible outside of Vagrant

Our Vagrantfile configures Ansible to make the process seamless, but there are a few differences when using Ansible outside of the context of a Vagrant machine.

Namely, the role that Vagrant uses is referred to by the folder name "girder" because you happen to have a working copy of Girder checked out, but this isn't required. By specifying the namespaced Ansible Galaxy version of Girder in your playbook and requirements file, the role will be fetched automatically.

Using Ansible to configure a running Girder instance

The Girder role also provides a fully fledged Ansible client to configure Girder in a declarative manner.

For details on usage, see the documentation on the Girder Ansible client.

See also:

The girder-configure-lib example demonstrates usage of the Girder Ansible client.

FAQ

How do I control the user that Girder runs under?

The Ansible playbook assumes that the user being used to provision the machine is the user which Girder will run as. This greatly simplifies the role logic and reduces problematic behavior with privilege deescalation in Ansible.

See http://docs.ansible.com/ansible/become.html#becoming-an-unprivileged-user for more information.

3.1.6 SFTP Service

In addition to its normal HTTP server, the Girder package also includes a lightweight SFTP server that can be used to serve the same underlying data using the SSH File Transfer Protocol. This server provides a read-only view of the data in a Girder instance, and supports either anonymous access (users must pass the username "anonymous" in their SFTP client), or authenticated access with their Girder login name and password. This protocol can make it much easier to download large nested datasets with many individual files, and is tolerant to network failure since it supports resuming interrupted downloads of entire data hierarchies.

After installing the Girder package via pip, you should now see the `girder-sftpd` executable in your PATH. Running it will start the SFTP server using the same database configuration as the main Girder HTTP server.

The SFTP server requires a private key file for secure communication with clients. If you do not pass an explicit path to an RSA private key file, the service will look for one at `~/.ssh/id_rsa`. It's recommended to make a special key just for the SFTP service, e.g.:

```
ssh-keygen -t rsa -b 2048 -f my_key.rsa -N ''
girder-sftpd -i my_key.rsa
```

You can control the port on which the server binds by passing a `-p <port>` argument to the server CLI. The default port is 8022.

3.2 User Documentation

3.2.1 User Guide

Girder is a Data Management Toolkit. It is a complete back-end (server side) technology that can be used with other applications via its RESTful API, or it can be used via its own front-end (client side web pages and JavaScript).

Girder is designed to be robust, fast, scalable, extensible, and easy to understand.

Girder is built in Python.

Girder is open source, licensed under the Apache License, Version 2.0.

Document Conventions

This User Guide is written for end-users of Girder, rather than developers. If you have suggestions or questions about this documentation, feel free to contact us on GitHub, the mailing list or, through Kitware support.

Girder specific entities will be `formatted like this`.

Concepts

Users

Like in many systems, `Users` in Girder correspond to the identity of a user of the system. It is possible to use many features of Girder anonymously (that is, without being logged in as a registered user), but typically in order to make changes to the system, a user must be logged in as their corresponding `User` account. `Users` can be granted permissions on resources in the system directly, and can belong to `Groups`.

Groups

Groups group together Users. Users can belong to any number of Groups, and usually join by being invited and accepting the invitation. One of the main purposes of Groups is to allow role-based access control; resources can grant access to Groups rather than just individual users, such that changing access to sets of resources can be managed simply by changing Group membership. See the *Permissions* section for more information about group-based access control.

Collections

Collections are the top level objects in the data organization hierarchy. Within each Collection, there can be many Folders, and the Collection itself is also an access controlled resource. Typically Collections are used to group data that share something in common, such as what project the data are used for, or what institution they belong to.

Folders

A Girder Folder is the common software concept of a folder, namely a hierarchically nested organizational structure. Girder Folders can contain nothing (although this may not be particularly useful), other Folders, Items, or a combination of Folders and Items. Folders in Girder have permissions set on them, and the Items within them inherit permissions from their containing Folders.

Items

A Girder Item is the basic unit of data in the system. Items live beneath Folders and contain 0 or more Files. Items in Girder do not have permissions set on them, they inherit permissions by virtue of living in a Folder (which has permissions set on it). Most Items contain a single File, except in special cases where multiple files make up a single piece of data.

Each Item may contain any number of arbitrary key/value pairs, termed metadata. Metadata keys must be non-empty strings and must not contain a period ('.') or begin with a dollar sign ('$'). Metadata values can be anything, including strings, numeric values, and even arbitrary JSON objects.

Files

Files represent raw data objects, just like the typical concept of files in a filesystem. Files exist within Items, typically with a one-to-one relationship between the File and its containing Item. Files in Girder are much like files on a filesystem, but they are actually more abstract. For instance, some Files are simply links to external URLs. All Files that are not external links must be contained within an Assetstore.

Assetstores

Assetstores are an abstraction representing a repository where the raw bytes of Files are actually stored. The Assetstores known to a Girder instance may only be set up and managed by administrator Users.

In the core of Girder, there are three supported Assetstore types:

- **Filesystem**

Files uploaded into this type of `Assetstore` will be stored on the local system filesystem of the server using content-addressed storage. Simply specify the root directory under which files should be stored.

Note: If your Girder environment has multiple different application servers and you plan to use the Filesystem assetstore type, you must set the assetstore root to a location on the filesystem that is shared between all of the application servers.

- **GridFS**

This `Assetstore` type stores files directly within your Mongo database using the **GridFS** model. You must specify the database name where files will be stored; for now, the same credentials will be used for this database as for the main application database.

This database type has the advantage of automatically scaling horizontally with your DBMS. However, it is marginally slower than the Filesystem assetstore type in a typical single-server use case.

- **S3**

This `Assetstore` type stores files in an **Amazon S3** bucket. You must provide the bucket name, an optional path prefix within the bucket, and authentication credentials for the bucket. When using this assetstore type, Girder acts as the broker for the data within S3 by authorizing the user agent via signed HTTP requests. The primary advantage of this type of assetstore is that none of the actual bytes of data being uploaded and downloaded ever go through the Girder system, but instead are sent directly between the client and S3.

If you want to use an S3 assetstore, the bucket used must support CORS requests. This can be edited by navigating to the bucket in the AWS S3 console, selecting **Properties**, then **Permissions**, and then clicking **Edit CORS Configuration**. The below CORS configuration is sufficient for Girder's needs:

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>PUT</AllowedMethod>
        <AllowedMethod>POST</AllowedMethod>
        <MaxAgeSeconds>3000</MaxAgeSeconds>
        <ExposeHeader>ETag</ExposeHeader>
        <AllowedHeader>*</AllowedHeader>
    </CORSRule>
</CORSConfiguration>
```

Permissions

Permission Levels There are four levels of permission a `User` can have on a resource. These levels are in a strict hierarchy with a higher permission level including all of the permissions below it. The levels are:

1. No permission (cannot view, edit, or delete a resource)
2. `READ` permission (can view and download resources)
3. `WRITE` permission (includes `READ` permission, can edit the properties of a resource)
4. `ADMIN` permission (includes `READ` and `WRITE` permission, can delete the resource and also control access on it)

A site administrator always has permission to take any action.

Permission Model Permissions are resolved at the level of a User, i.e., for any User, an attempt to take a certain action will be allowed or disallowed based on the permissions for that User, as a function of the resource, the operation, the permissions set on that resource for that User, and the permissions set on that resource by any Groups the User is a member of.

Permissions are always additive. That is, given a User with a certain permission on a resource, that permission can not be taken away from the User by addition of other permissions to the system, but only through removing existing permissions to that User or removing that User from a Group. Once again, a site admin always has permission to take any action.

Collections Collections can be Public (meaning viewable even by anonymous users) or Private (meaning viewable only by those with READ access). Collections can have permissions set on them at the individual User level and Group level, meaning that a given User or Group can have READ, WRITE, or ADMIN permissions set on the Collection.

Folders Folders can be Public (meaning viewable even by anonymous users) or Private (meaning viewable only by those with READ access). Folders can have permissions set on them at the individual User level and Group level, meaning that a given User or Group can have READ, WRITE, or ADMIN permissions set on the Folder. Folders inherit permissions from their parent Folder.

Items Items always inherit their permissions from their parent Folder. Each access-controlled resource (e.g., Folder, Collection) has a list of permissions granted on it, and each item in that list is a mapping of either Users to permission level or Groups to permission level. This is best visualized by opening the "Access control" dialog on a Folder in the hierarchy. The actual permission level that a User has on that resource is defined as: the maximum permission level available based on the permissions granted to any Groups that the User is member of, or permissions granted to that User specifically.

Groups For access control, Groups can be given any level of access to a resource that an individual User can, and this is managed at the level of the resource in question.

For permissions on Groups themselves, Public Groups are viewable (READ permission) to anyone, even anonymous users. Private Groups are not viewable or even listable to any Users except those that are members of the Group, or those that have been invited to the Group.

Groups have three levels of roles that Users can have within the Group. They can be Members, Moderators (also indicates that they are Members), and Administrators (also indicates that they are Members).

Users that are not Members of a group can request to become Members of a Group if that Group is Public.

Members of a Group can see the membership list of the Group, including roles, and can see pending requests and invitations for the group. If a User has been invited to a Group, they have Member access to the Group even before they have accepted the invitation. A Member of a Group can leave the group, at which point they are no longer Members of the Group.

Moderators of a Group have all of the abilities of Group Members. Moderators can also invite Users to become Members, can accept or reject a request by a User to become a Member, can remove Members or Moderators from the Group, and can edit the Group which includes changing the name and description and changing the Public/Private status of the Group.

Administrators of a Group have all of the abilities of Group Moderators. Administrators can also delete the Group, promote a Member to Moderator or Administrator, demote an Administrator or Moderator to Member, and remove any Member, Moderator, or Administrator from the Group.

The creator of a Group is an Administrator of a group. Any logged in User can create a Group.

User `Users` have `ADMIN` access on themselves, and have `READ` access on other `Users`.

API keys

Like many web services, Girder's API is designed for programmatic interaction. API keys can facilitate these sorts of interactions – they enable client applications to interact with the server on behalf of your user without actually authenticating with your password. They can also be granted restricted access to only a limited set of functionality of the API.

Under the **My account** page, there is a tab called **API keys** where these keys can be created and managed. You can have many API keys; in fact, it's recommended to use a different key for each different client application that needs authenticated access to the Girder server. By convention, the **Name** field of API keys can be used to specify what application is making use of the key in a human-readable way, although you may name your keys however you want.

Each API key can be used to gain authentication tokens just like when you log in with a username and password. If you want to limit the maximum amount of time that these tokens last, you can do so on a per-key basis, or leave the token duration field empty to use the server default.

When creating and updating API keys, you can also select among two modes: you can either grant full access to the API key, which gives unrestricted API access as though you are logged in as your user, or you can choose limited functionality scopes from a list of checkboxes to restrict the sorts of actions that the key will allow.

It is also possible to deactivate a key temporarily. If you deactivate an existing key, it will immediately delete all active tokens created with that key, and also stop that key from being able to create new tokens until you activate it once again. Alternatively, you can delete the key altogether, which will make the key and any tokens created with it never work again.

3.3 Developer Documentation

3.3.1 API Documentation

RESTful API

Clients access Girder servers uniformly via its RESTful web API. By providing a single, stable, consistent web API, it is possible to write multiple interchangeable clients using different technologies.

When a Girder instance is deployed, it typically also serves a page that uses Swagger to document all available RESTful endpoints in the web API and also provide an easy way for users to execute those endpoints with parameters of their choosing. In this way, the Swagger page is just the simplest and lightest client application for Girder. This page is served out of the path `/api` under the root path of your Girder instance.

Models

In Girder, the model layer is responsible for actually interacting with the underlying database. Model classes are where the documents representing resources are actually saved, retrieved, and deleted from the DBMS. Validation of the resource documents is also done in the model layer, and is invoked each time a document is about to be saved.

Typically, there is a model class for each resource type in the system. These models are loaded as singletons for efficiency, and can be accessed in REST resources or other models by invoking `self.model('foo')`, where `foo` is the name of the model. For example:

```
groups = self.model('group').list(user=self.getCurrentUser())
```

All models that require the standard access control semantics should extend the *AccessControlledModel* class. Otherwise, they should extend the *Model* class.

All model classes must have an `initialize` method in which they declare the name of their corresponding Mongo collection, as well as any collection indices they require. For example:

```
from girder.models.model_base import Model

class Cat(Model):
    def initialize(self):
        self.name = 'cat_collection'
```

The above model singleton could then be accessed via:

```
self.model('cat')
```

If you wish to use models in something other than a REST Resource or Model, either mixin or instantiate the *ModelImporter* class.

Model Helper Functions

girder.models.**getDbConfig**()
Get the database configuration values from the cherrypy config.

girder.models.**getDbConnection**(*uri=None*, *replicaSet=None*, *autoRetry=True*, ***kwargs*)
Get a MongoClient object that is connected to the configured database. We lazy-instantiate a module-level singleton, the MongoClient objects manage their own connection pools internally. Any extra kwargs you pass to this method will be passed through to the MongoClient.

Parameters

- **uri** – if specified, connect to this mongo db rather than the one in the config.
- **replicaSet** – if uri is specified, use this replica set.
- **autoRetry** (*bool*) – if this connection should automatically retry operations in the event of an AutoReconnect exception. If you're testing the connection, set this to False. If disabled, this also will not cache the mongo client, so make sure to only disable if you're testing a connection.

Model Base

class girder.models.model_base.**AccessControlledModel**
Any model that has access control requirements should inherit from this class. It enforces permission checking in the load() method and provides convenient methods for testing and requiring user permissions. It also provides methods for setting access control policies on the resource.

copyAccessPolicies(*src*, *dest*, *save=False*)
Copies the set of access control policies from one document to another.

Parameters

- **src** (*dict*) – The source document to copy policies from.
- **dest** (*dict*) – The destination document to copy policies onto.
- **save** (*bool*) – Whether to save the destination document after copying.

Returns The modified destination document.

filter(*doc*, *user*, *additionalKeys=None*)
Filter this model for the given user according to the user's access level. Also adds the special _accessLevel field to the document to indicate the user's highest access level. This filters a single document that the user has at least read access to. For filtering a set of documents, see filterResultsByPermission().

Parameters

- **doc** (*dict or None*) – The document of this model type to be filtered.
- **user** (*dict or None*) – The current user for whom we are filtering.
- **additionalKeys** (*list, tuple, or None*) – Any additional keys that should be included in the document for this call only.

Returns The filtered document (dict).

filterResultsByPermission(*cursor*, *user*, *level*, *limit=0*, *offset=0*, *removeKeys=()*)
Given a database result cursor, this generator will yield only the results that the user has the given level of access on, respecting the limit and offset specified.

Parameters

- **cursor** – The database cursor object from "find()".
- **user** (*dict or None*) – The user to check policies against.
- **level** (AccessType) – The access level.
- **limit** (*int*) – Maximum number of documents to return
- **offset** (*int*) – The offset into the results
- **removeKeys** (list) – List of keys that should be removed from each matching document.

getAccessLevel(*doc*, *user*)
Return the maximum access level for a given user on a given object. This can be useful for alerting the user which set of actions they are able to perform on the object in advance of trying to call them.

Parameters

- **doc** – The object to check access on.
- **user** – The user to get the access level for.

Returns The max AccessType available for the user on the object.

getFullAccessList(*doc*)
Return an object representing the full access list on this document. This simply includes the names of the users and groups with the ACL.

If the document contains references to users or groups that no longer exist, they are simply removed from the ACL, and the modified ACL is persisted at the end of this method if any removals occurred.

Parameters **doc** (*dict*) – The document whose ACL to return.

Returns A dict containing *users* and *groups* keys.

hasAccess(*doc*, *user=None*, *level=0*)
This method looks through the object's permission set and determines whether the user has the given permission level on the object.

Parameters

- **doc** (*dict*) – The document to check permission on.
- **user** (*dict*) – The user to check against.

- **level** (AccessType) – The access level.

Returns Whether the access is granted.

list(*user=None*, *limit=0*, *offset=0*, *sort=None*)
Return a list of documents that are visible to a user.

Parameters

- **user** (*dict or None*) – The user to filter for
- **limit** (*int*) – Maximum number of documents to return
- **offset** (*int*) – The offset into the results
- **sort** (*List of (key, order) tuples*) – The sort order

load(*id*, *level=2*, *user=None*, *objectId=True*, *force=False*, *fields=None*, *exc=False*)
Override of Model.load to also do permission checking.

Parameters

- **id** (*str or ObjectId*) – The id of the resource.
- **level** (AccessType) – The required access type for the object.
- **user** (*dict or None*) – The user to check access against.
- **objectId** (*bool*) – Whether the id should be coerced to ObjectId type.
- **force** (*bool*) – If you explicitly want to circumvent access checking on this resource, set this to True.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **exc** (*bool*) – If not found, throw a ValidationException instead of returning None.

Raises **ValidationException** – If an invalid ObjectId is passed.

Returns The matching document, or None if no match exists.

prefixSearch(*query*, *user=None*, *filters=None*, *limit=0*, *offset=0*, *sort=None*, *fields=None*, *level=0*)
Custom override of Model.prefixSearch to also force permission-based filtering. The parameters are the same as Model.prefixSearch.

Parameters

- **query** (*str*) – The prefix string to look for
- **user** (*dict or None*) – The user to apply permission filtering for.
- **filters** (*dict*) – Any additional query operators to apply.
- **limit** (*int*) – Maximum number of documents to return
- **offset** (*int*) – The offset into the results
- **sort** (*List of (key, order) tuples.*) – The sort order.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **level** (girder.constants.AccessType) – The access level to require.

Returns A pymongo cursor. It is left to the caller to build the results from the cursor.

requireAccess(*doc*, *user=None*, *level=0*)
This wrapper just provides a standard way of throwing an access denied exception if the access check fails.

setAccessList(*doc*, *access*, *save=False*)
Set the entire access control list to the given value. This also saves the resource in its new state to the database.

Parameters

- **doc** (*dict*) – The resource to update.
- **access** (*dict*) – The new access control list to set on the object.
- **save** (*boolean*) – Whether to save after updating.

Returns The updated resource.

setGroupAccess(*doc*, *group*, *level*, *save=False*)
Set group-level access on the resource.

Parameters

- **doc** (*dict*) – The resource document to set access on.
- **group** (*dict*) – The group to grant or remove access to.
- **level** (*AccessType or None*) – What level of access the group should have. Set to None to remove all access for this group.
- **save** (*bool*) – Whether to save the object to the database afterward. Set this to False if you want to wait to save the document for performance reasons.

Returns The updated resource document.

setPublic(*doc*, *public*, *save=False*)
Set the flag for public read access on the object.

Parameters

- **doc** (*dict*) – The document to update permissions on.
- **public** (*bool*) – Flag for public read access.
- **save** (*bool*) – Whether to save the object to the database afterward. Set this to False if you want to wait to save the document for performance reasons.

Returns The updated resource document.

setUserAccess(*doc*, *user*, *level*, *save=False*)
Set user-level access on the resource.

Parameters

- **doc** (*dict*) – The resource document to set access on.
- **user** (*dict*) – The user to grant or remove access to.
- **level** (*AccessType or None*) – What level of access the user should have. Set to None to remove all access for this user.
- **save** (*bool*) – Whether to save the object to the database afterward. Set this to False if you want to wait to save the document for performance reasons.

Returns The modified resource document.

textSearch(*query*, *user=None*, *filters=None*, *limit=0*, *offset=0*, *sort=None*, *fields=None*, *level=0*)
Custom override of Model.textSearch to also force permission-based filtering. The parameters are the same as Model.textSearch.

Parameters

- **query** (*str*) – The text query. Will be stemmed internally.
- **user** (*dict or None*) – The user to apply permission filtering for.
- **filters** (*dict*) – Any additional query operators to apply.
- **limit** (*int*) – Maximum number of documents to return
- **offset** (*int*) – The offset into the results
- **sort** (*List of (key, order) tuples*) – The sort order
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **level** (girder.constants.AccessType) – The access level to require.

exception girder.models.model_base.**AccessException**(*message*, *extra=None*)
Represents denial of access to a resource.

exception girder.models.model_base.**GirderException**(*message*, *identifier=None*)
Represents a general exception that might occur in regular use. From the user perspective, these are failures, but not catastrophic ones. An identifier can be passed, which allows receivers to check the exception without relying on the text of the message. It is recommended that identifiers are a dot-separated string consisting of the originating python module and a distinct error. For example, 'girder.model.assetstore.no-current-assetstore'.

class girder.models.model_base.**Model**
Model base class. Models are responsible for abstracting away the persistence layer. Each collection in the database should have its own model. Methods that deal with database interaction belong in the model layer.

ensureIndex(*index*)
Like ensureIndices, but declares just a single index rather than a list of them.

ensureIndices(*indices*)
Subclasses should call this with a list of strings representing fields that should be indexed in the database if there are any. Otherwise, it is not necessary to call this method. Elements of the list may also be a list or tuple, where the second element is a dictionary that will be passed as kwargs to the pymongo create_index call.

ensureTextIndex(*index*, *language='english'*)
Call this during initialize() of the subclass if you want your model to have a full-text searchable index. Each collection may have zero or one full-text index.

Parameters **language** (*str*) – The default_language value for the text index, which is used for stemming and stop words. If the text index should not use stemming and stop words, set this param to 'none'.

exposeFields(*level*, *fields*)
Expose model fields to users with the given access level. Subclasses should call this in their initialize method to declare what fields should be exposed to what access levels if they are using the default filter implementation in this class. Since filtered fields are sets, this method is idempotent.

Parameters

- **level** (AccessType) – The required access level for the field.

- **fields** (*str, list, or tuple*) – A field or list of fields to expose for that level.

filter(*doc*, *user=None*, *additionalKeys=None*)

Filter this model for the given user. This is a default implementation that assumes this model has no notion of access control, and simply allows all keys under READ access level, and conditionally allows any keys assigned to SITE_ADMIN level.

Parameters

- **doc** (*dict or None*) – The document of this model type to be filtered.
- **user** (*dict or None*) – The current user for whom we are filtering.
- **additionalKeys** (*list, tuple, set, or None*) – Any additional keys that should be included in the document for this call only.

Returns The filtered document (dict).

filterDocument(*doc*, *allow=None*)

This method will filter the given document to make it suitable to output to the user.

Parameters

- **doc** (*dict*) – The document to filter.
- **allow** (*List of strings*) – The whitelist of fields to allow in the output document.

find(*query=None*, *offset=0*, *limit=0*, *timeout=None*, *fields=None*, *sort=None*, ***kwargs*)

Search the collection by a set of parameters. Passes any extra kwargs through to the underlying pymongo.collection.find function.

Parameters

- **query** (*dict*) – The search query (see general MongoDB docs for "find()")
- **offset** (*int*) – The offset into the results
- **limit** (*int*) – Maximum number of documents to return
- **timeout** (*int*) – Cursor timeout in ms. Default is no timeout.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **sort** (*List of (key, order) tuples.*) – The sort order.

Returns A pymongo database cursor.

findOne(*query=None*, *fields=None*, ***kwargs*)

Search the collection by a set of parameters. Passes any kwargs through to the underlying pymongo.collection.find_one function.

Parameters

- **query** (*dict*) – The search query (see general MongoDB docs for "find()")
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **sort** (*List of (key, order) tuples.*) – The sort order.

Returns the first object that was found, or None if none found.

hideFields(*level*, *fields*)
Hide a field, i.e. make sure it is not exposed via the default filtering method. Since the filter uses a white list, it is only ever necessary to call this for fields that were added previously with exposeFields().

Parameters

- **level** (AccessType) – The access level to remove the fields from.
- **fields** (*str, list, or tuple*) – The field or fields to remove from the white list.

increment(*query*, *field*, *amount*, ***kwargs*)
This is a specialization of the update method that atomically increments a field by a given amount. Additional kwargs are passed directly through to update.

Parameters

- **query** (*dict*) – The search query for documents to update, see general MongoDB docs for "find()"
- **field** (*str*) – The name of the field in the document to increment.
- **amount** (*int or float*) – The amount to increment the field by.

initialize()
Subclasses should override this and set the name of the collection as self.name. Also, they should set any indexed fields that they require.

load(*id*, *objectId=True*, *fields=None*, *exc=False*)
Fetch a single object from the database using its _id field.

Parameters

- **id** (*string or ObjectId*) – The value for searching the _id field.
- **objectId** (*bool*) – Whether the id should be coerced to ObjectId type.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **exc** (*bool*) – Whether to raise a ValidationException if there is no document with the given id.

Returns The matching document, or None.

prefixSearch(*query*, *offset=0*, *limit=0*, *sort=None*, *fields=None*, *filters=None*, *prefixSearchFields=None*, ***kwargs*)
Search for documents in this model's collection by a prefix string. The fields that will be searched based on this prefix must be set as the `prefixSearchFields` attribute of this model, which must be an iterable. Elements of this iterable must be either a string representing the field name, or a 2-tuple in which the first element is the field name, and the second element is a string representing the regex search options.

Parameters

- **query** (*str*) – The prefix string to look for
- **offset** (*int*) – The offset into the results
- **limit** (*int*) – Maximum number of documents to return
- **sort** (*List of (key, order) tuples.*) – The sort order.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion*

or exclusion projection.) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.

- **filters** (*dict*) – Any additional query operators to apply.
- **prefixSearchFields** – To override the model's prefixSearchFields attribute for this invocation, pass an alternate iterable.

Returns A pymongo cursor. It is left to the caller to build the results from the cursor.

reconnect()
Reconnect to the database and rebuild indices if necessary. Users should typically not have to call this method.

remove(*document*, ***kwargs*)
Delete an object from the collection; must have its _id set.

Parameters **document** – the item to remove.

removeWithQuery(*query*)
Remove all documents matching a given query from the collection. For safety reasons, you may not pass an empty query.

Parameters **query** (*dict*) – The search query for documents to delete, see general MongoDB docs for "find()"

save(*document*, *validate=True*, *triggerEvents=True*)
Create or update a document in the collection. This triggers two events; one prior to validation, and one prior to saving. Either of these events may have their default action prevented.

Parameters

- **document** (*dict*) – The document to save.
- **validate** (*bool*) – Whether to call the model's validate() before saving.
- **triggerEvents** – Whether to trigger events for validate and pre- and post-save hooks.

subtreeCount(*doc*)
Return the size of the subtree rooted at the given document. In general, if this contains items or folders, it will be the count of the items and folders in all containers. If it does not, it will be 1. This returns the absolute size of the subtree, it does not filter by permissions.

Parameters **doc** (*dict*) – The root of the subtree.

textSearch(*query*, *offset=0*, *limit=0*, *sort=None*, *fields=None*, *filters=None*, ***kwargs*)
Perform a full-text search against the text index for this collection.

Parameters

- **query** (*str*) – The text query. Will be stemmed internally.
- **offset** (*int*) – The offset into the results
- **limit** (*int*) – Maximum number of documents to return
- **sort** (*List of (key, order) tuples.*) – The sort order.
- **fields** (*str, list of strings or tuple of strings for fields to be included from the document, or dict for an inclusion or exclusion projection.*) – A mask for filtering result documents by key, or None to return the full document, passed to MongoDB find() as the *projection* param.
- **filters** (*dict*) – Any additional query operators to apply.

Returns A pymongo cursor. It is left to the caller to build the results from the cursor.

update (*query*, *update*, *multi=True*)
This method should be used for updating multiple documents in the collection. This is useful for things like removing all references in this collection to a document that is being deleted from another collection.

For updating a single document, use the save() model method instead.

Parameters

- **query** (*dict*) – The search query for documents to update, see general MongoDB docs for "find()"
- **update** (*dict*) – The update specifier.
- **multi** (*bool*) – Whether to update a single document, or all matching documents.

Returns A pymongo UpdateResult object.

validate (*doc*)
Models should implement this to validate the document before it enters the database. It must return the document with any necessary filters applied, or throw a ValidationException if validation of the document fails.

Parameters **doc** (*dict*) – The document to validate before saving to the collection.

exception girder.models.model_base.**ValidationException** (*message*, *field=None*)
Represents validation failure in the model layer. Raise this with a message and an optional field property. If one of these is thrown in the model during a REST request, it will respond as a 400 status.

User

class girder.models.user.**User**
This model represents the users of the system.

adminApprovalRequired (*user*)
Returns True if the registration policy requires admin approval and this user is pending approval.

authenticate (*login*, *password*)
Validate a user login via username and password. If authentication fails, a AccessException is raised.

Parameters

- **login** (*str*) – The user's login or email.
- **password** (*str*) – The user's password.

Returns The corresponding user if the login was successful.

Return type dict

canLogin (*user*)
Returns True if the user is allowed to login, e.g. email verification is not needed and admin approval is not needed.

countFolders (*user*, *filterUser=None*, *level=None*)
Returns the number of top level folders under this user. Access checking is optional; to circumvent access checks, pass level=None.

Parameters

- **user** – The user whose top level folders to count.
- **filterUser** (*dict or None*) – If performing access checks, the user to check against.

- **level** – The required access level, or None to return the raw top-level folder count.

createUser(*login*, *password*, *firstName*, *lastName*, *email*, *admin=False*, *public=True*)
Create a new user with the given information. The user will be created with the default "Public" and "Private" folders.

Parameters

- **admin** (*bool*) – Whether user is global administrator.
- **public** (*bool*) – Whether user is publicly visible.

Returns The user document that was created.

emailVerificationRequired(*user*)
Returns True if email verification is required and this user has not yet verified their email address.

fileList(*doc*, *user=None*, *path=''*, *includeMetadata=False*, *subpath=True*, *data=True*)
This function generates a list of 2-tuples whose first element is the relative path to the file from the user's folders root and whose second element depends on the value of the *data* flag. If *data=True*, the second element will be a generator that will generate the bytes of the file data as stored in the assetstore. If *data=False*, the second element is the file document itself.

Parameters

- **doc** – the user to list.
- **user** – a user used to validate data that is returned.
- **path** – a path prefix to add to the results.
- **includeMetadata** – if True and there is any metadata, include a result which is the JSON string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.
- **subpath** – if True, add the user's name to the path.
- **data** (*bool*) – If True return raw content of each file as stored in the assetstore, otherwise return file document.

getAdmins()
Helper to return a cursor of all site-admin users. The number of site admins is assumed to be small enough that we will not need to page the results for now.

remove(*user*, *progress=None*, ***kwargs*)
Delete a user, and all references to it in the database.

Parameters

- **user** (*dict*) – The user document to delete.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – A progress context to record progress on.

search(*text=None*, *user=None*, *limit=0*, *offset=0*, *sort=None*)
List all users. Since users are access-controlled, this will filter them by access policy.

Parameters

- **text** – Pass this to perform a full-text search for users.
- **user** – The user running the query. Only returns users that this user can see.
- **limit** – Result limit.
- **offset** – Result offset.

- **sort** – The sort structure to pass to pymongo.

Returns Iterable of users.

setPassword(*user*, *password*, *save=True*)
Change a user's password.

Parameters

- **user** – The user whose password to change.
- **password** – The new password. If set to None, no password will be stored for this user. This should be done in cases where an external system is responsible for authenticating the user.

subtreeCount(*doc*, *includeItems=True*, *user=None*, *level=None*)
Return the size of the user's folders. The user is counted as well.

Parameters

- **doc** – The user.
- **includeItems** (*bool*) – Whether to include items in the subtree count, or just folders.
- **user** – If filtering by permission, the user to filter against.
- **level** (*AccessLevel*) – If filtering by permission, the required permission level.

updateSize(*doc*)
Recursively recomputes the size of this user and its underlying folders and fixes the sizes as needed.

Parameters **doc** (*dict*) – The user.

validate(*doc*)
Validate the user every time it is stored in the database.

Password

class girder.models.password.**Password**
This model deals with managing user passwords.

authenticate(*user*, *password*)
Authenticate a user.

Parameters

- **user** (*dict*) – The user document.
- **password** (*str*) – The attempted password.

Returns Whether authentication succeeded (bool).

encryptAndStore(*password*)
Encrypt and store the given password. The exact internal details and mechanisms used for storage are abstracted away, but the guarantee is made that once this method is called on a password and the returned salt and algorithm are stored with the user document, calling Password.authenticate() with that user document and the same password will return True.

Parameters **password** (*str*) – The password to encrypt and store.

Returns {tuple} (salt, hashAlg) The salt to store with the user document and the algorithm used for secure storage. Both should be stored in the corresponding user document as 'salt' and 'hashAlg' respectively.

hasPassword(*user*)
Returns whether or not the given user has a password stored in the database. If not, it is expected that the user will be authenticated by an external service.

Parameters **user** (*dict*) – The user to test.

Returns bool

Token

class girder.models.token.**Token**
This model stores session tokens for user authentication.

addScope(*token*, *scope*)
Add a scope to this token. If the token already has the scope, this is a no-op.

clearForApiKey(*apiKey*)
Delete all tokens corresponding to an API key.

createToken(*user=None*, *days=None*, *scope=None*, *apiKey=None*)
Creates a new token. You can create an anonymous token (such as for CSRF mitigation) by passing "None" for the user argument.

Parameters

- **user** (*dict*) – The user to create the session for.
- **days** (*float or int*) – The lifespan of the session in days. If not passed, uses the database setting for cookie lifetime.
- **scope** (*str or list of str*) – Scope or list of scopes this token applies to. By default, will create a user authentication token.
- **apiKey** (*dict*) – If this token is being created via an API key, pass it so that we can record the provenance for cleanup and auditing.

Returns The token document that was created.

getAllowedScopes(*token*)
Return the list of allowed scopes for a given token.

hasScope(*token*, *scope*)
Test whether the given token has the given set of scopes. Use this rather than comparing manually, since this method is backward compatible with tokens that do not contain a scope field.

Parameters

- **token** (*dict*) – The token object.
- **scope** (*str or list of str*) – A scope or set of scopes that will be tested as a subset of the given token's allowed scopes.

Group

class girder.models.group.**Group**
Groups are simply groups of users. The primary use of grouping users is to simplify access control for resources in the system, but they can be used for other purposes that require groupings of users as well.

Group membership is stored in the database on the user document only; there is no "users" field in this model. This is to optimize for the most common use case for querying membership, which involves checking access control policies, which is always done relative to a specific user. The task of querying all members within a

group is much less common and typically only performed on a single group at a time, so doing a find on the indexed group list in the user collection is sufficiently fast.

Users with READ access on the group can see the group and its members. Users with WRITE access on the group can add and remove members and change the name or description. Users with ADMIN access can promote group members to grant them WRITE or ADMIN access, and can also delete the entire group.

This model uses a custom implementation of the access control methods, because it uses only a subset of its capabilities and provides a more optimized implementation for that subset. Specifically: read access is implied by membership in the group or having an invitation to join the group, so we don't store read access in the access document as normal. Another constraint is that write and admin access on the group can only be granted to members of the group. Also, group permissions are not allowed on groups for the sake of simplicity.

addUser (*group*, *user*, *level=0*)
Add the user to the group. Records membership in the group in the user document, and also grants the specified access level on the group itself to the user. Any group member has at least read access on the group. If the user already belongs to the group, this method can be used to change their access level within it.

createGroup (*name*, *creator*, *description=''*, *public=True*)
Create a new group. The creator will be given admin access to it.

Parameters

- **name** (*str*) – The name of the folder.
- **description** (*str*) – Description for the folder.
- **public** (*bool*) – Whether the group is publicly visible.
- **creator** (*dict*) – User document representing the creator of the group.

Returns The group document that was created.

getAccessLevel (*doc*, *user*)
Return the maximum access level for a given user on the group.

Parameters

- **doc** – The group to check access on.
- **user** – The user to get the access level for.

Returns The max AccessType available for the user on the object.

getFullRequestList (*group*)
Return the set of all outstanding requests, filled in with the login and full names of the corresponding users.

Parameters **group** (*dict*) – The group to get requests for.

getInvites (*group*, *limit=0*, *offset=0*, *sort=None*)
Return a page of outstanding invitations to a group. This is simply a list of users invited to the group currently.

Parameters

- **group** – The group to find invitations for.
- **limit** – Result set size limit.
- **offset** – Offset into the results.
- **sort** – The sort field.

getMembers (*group*, *offset=0*, *limit=0*, *sort=None*)
Return the list of all users who belong to this group.

Parameters

- **group** – The group to list members on.
- **offset** – Offset into the result set of users.
- **limit** – Result set size limit.
- **sort** – Sort parameter for the find query.

Returns List of user documents.

hasAccess(*doc*, *user=None*, *level=0*)
This overrides the default AccessControlledModel behavior for checking access to perform an optimized subset of the access control behavior.

Parameters

- **doc** (*dict*) – The group to check permission on.
- **user** (*dict*) – The user to check against.
- **level** (AccessType) – The access level.

Returns Whether the access is granted.

inviteUser(*group*, *user*, *level=0*)
Invite a user to join the group. Inviting them automatically grants the user read access to the group so that they can see it. Once they accept the invitation, they will be given the specified level of access.

If the user has requested an invitation to this group, calling this will accept their request and add them to the group at the access level specified.

joinGroup(*group*, *user*)
This method either accepts an invitation to join a group, or if the given user has not been invited to the group, this will create an invitation request that moderators and admins may grant or deny later.

listMembers(*group*, *offset=0*, *limit=0*, *sort=None*)
List members of the group.

remove(*group*, ***kwargs*)
Delete a group, and all references to it in the database.

Parameters **group** (*dict*) – The group document to delete.

removeUser(*group*, *user*)
Remove the user from the group. If the user is not in the group but has an outstanding invitation to the group, the invitation will be revoked. If the user has requested an invitation, calling this will deny that request, thereby deleting it.

setUserAccess(*doc*, *user*, *level*, *save=False*)
This override is used because we only need to augment the access field in the case of WRITE access and above since READ access is implied by membership or invitation.

updateGroup(*group*)
Updates a group.

Parameters **group** (*dict*) – The group document to update

Returns The group document that was edited.

Collection

class girder.models.collection.**Collection**

Collections are the top level roots of the data hierarchy. They are used to group and organize data that is meant to be shared amongst users.

countFolders(*collection*, *user=None*, *level=None*)

Returns the number of top level folders under this collection. Access checking is optional; to circumvent access checks, pass `level=None`.

Parameters

- **collection** (*dict*) – The collection.
- **user** (*dict or None*) – If performing access checks, the user to check against.
- **level** – The required access level, or None to return the raw top-level folder count.

createCollection(*name*, *creator=None*, *description=''*, *public=True*, *reuseExisting=False*)

Create a new collection.

Parameters

- **name** (*str*) – The name of the collection. Must be unique.
- **description** (*str*) – Description for the collection.
- **public** (*bool*) – Public read access flag.
- **creator** (*dict*) – The user who is creating this collection.
- **reuseExisting** (*bool*) – If a collection with the given name already exists return that collection rather than creating a new one.

Returns The collection document that was created.

fileList(*doc*, *user=None*, *path=''*, *includeMetadata=False*, *subpath=True*, *mimeFilter=None*, *data=True*)

This function generates a list of 2-tuples whose first element is the relative path to the file from the collection's root and whose second element depends on the value of the *data* flag. If *data=True*, the second element will be a generator that will generate the bytes of the file data as stored in the assetstore. If *data=False*, the second element is the file document itself.

Parameters

- **doc** – the collection to list.
- **user** – a user used to validate data that is returned.
- **path** – a path prefix to add to the results.
- **includeMetadata** – if True and there is any metadata, include a result which is the JSON string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.
- **subpath** – if True, add the collection's name to the path.
- **mimeFilter** (*list or tuple*) – Optional list of MIME types to filter by. Set to None to include all files.
- **data** (*bool*) – If True return raw content of each file as stored in the assetstore, otherwise return file document.

hasCreatePrivilege(*user*)
Tests whether a given user has the authority to create collections on this instance. This is based on the collection creation policy settings. By default, only admins are allowed to create collections.

Parameters **user** – The user to test.

Returns bool

remove(*collection*, *progress=None*, ***kwargs*)
Delete a collection recursively.

Parameters

- **collection** (*dict*) – The collection document to delete.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – A progress context to record progress on.

setAccessList(*doc*, *access*, *save=False*, *recurse=False*, *user=None*, *progress=<girder.utility.progress.ProgressContext object>*, *setPublic=None*)
Overrides AccessControlledModel.setAccessList to add a recursive option. When *recurse=True*, this will set the access list on all subfolders to which the given user has ADMIN access level. Any subfolders that the given user does not have ADMIN access on will be skipped.

Parameters

- **doc** (*collection*) – The collection to set access settings on.
- **access** (*dict*) – The access control list.
- **save** (*bool*) – Whether the changes should be saved to the database.
- **recurse** (*bool*) – Whether this access list should be propagated to all folders underneath this collection.
- **user** – The current user (for recursive mode filtering).
- **progress** (*girder.utility.progress.ProgressContext*) – Progress context to update.
- **setPublic** (*bool or None*) – Pass this if you wish to set the public flag on the resources being updated.

subtreeCount(*doc*, *includeItems=True*, *user=None*, *level=None*)
Return the size of the folders within the collection. The collection is counted as well.

Parameters

- **doc** – The collection.
- **includeItems** (*bool*) – Whether items should be included in the count.
- **user** – If filtering by permission, the user to filter against.
- **level** (*AccessLevel*) – If filtering by permission, the required permission level.

updateCollection(*collection*)
Updates a collection.

Parameters **collection** (*dict*) – The collection document to update

Returns The collection document that was edited.

updateSize(*doc*)
Recursively recomputes the size of this collection and its underlying folders and fixes the sizes as needed.

Parameters **doc** (*dict*) – The collection.

Folder

class girder.models.folder.**Folder**

Folders are used to store items and can also store other folders in a hierarchical way, like a directory on a filesystem. Every folder has its own set of access control policies, but by default the access control list is inherited from the folder's parent folder, if it has one. Top-level folders are ones whose parent is a user or a collection.

childFolders(*parent*, *parentType*, *user=None*, *limit=0*, *offset=0*, *sort=None*, *filters=None*, ***kwargs*)

This generator will yield child folders of a user, collection, or folder, with access policy filtering. Passes any kwargs to the find function.

Parameters

- **parent** – The parent object.
- **parentType** (*'user', 'folder', or 'collection'*) – The parent type.
- **user** – The user running the query. Only returns folders that this user can see.
- **limit** – Result limit.
- **offset** – Result offset.
- **sort** – The sort structure to pass to pymongo.
- **filters** – Additional query operators.

childItems(*folder*, *limit=0*, *offset=0*, *sort=None*, *filters=None*, ***kwargs*)

Generator function that yields child items in a folder. Passes any kwargs to the find function.

Parameters

- **folder** – The parent folder.
- **limit** – Result limit.
- **offset** – Result offset.
- **sort** – The sort structure to pass to pymongo.
- **filters** – Additional query operators.

clean(*folder*, *progress=None*, ***kwargs*)

Delete all contents underneath a folder recursively, but leave the folder itself.

Parameters

- **folder** (*dict*) – The folder document to delete.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – A progress context to record progress on.

copyFolder(*srcFolder*, *parent=None*, *name=None*, *description=None*, *parentType=None*, *public=None*, *creator=None*, *progress=None*, *firstFolder=None*)

Copy a folder, including all child items and child folders.

Parameters

- **srcFolder** (*dict*) – the folder to copy.
- **parent** (*dict*) – The parent document. Must be a folder, user, or collection.
- **name** (*str*) – The name of the new folder. None to copy the original name.

- **description** (*str*) – Description for the new folder. None to copy the original description.
- **parentType** (*str*) – What type the parent is: ('folder' | 'user' | 'collection')
- **public** (*bool, None, or 'original'.*) – Public read access flag. None to inherit from parent, 'original' to inherit from original folder.
- **creator** (*dict*) – user representing the creator of the new folder.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – a progress context to record process on.
- **firstFolder** – if not None, the first folder copied in a tree of folders.

Returns the new folder document.

copyFolderComponents (*srcFolder*, *newFolder*, *creator*, *progress*, *firstFolder=None*)
Copy the items, subfolders, and extended data of a folder that was just copied.

Parameters

- **srcFolder** (*dict*) – the original folder.
- **newFolder** (*dict*) – the new folder.
- **creator** (*dict*) – user representing the creator of the new folder.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – a progress context to record process on.
- **firstFolder** – if not None, the first folder copied in a tree of folders.

Returns the new folder document.

countFolders (*folder*, *user=None*, *level=None*)
Returns the number of subfolders within the given folder. Access checking is optional; to circumvent access checks, pass `level=None`.

Parameters

- **folder** (*dict*) – The parent folder.
- **user** (*dict or None*) – If performing access checks, the user to check against.
- **level** – The required access level, or None to return the raw subfolder count.

countItems (*folder*)
Returns the number of items within the given folder.

createFolder (*parent*, *name*, *description=''*, *parentType='folder'*, *public=None*, *creator=None*, *allowRename=False*, *reuseExisting=False*)
Create a new folder under the given parent.

Parameters

- **parent** (*dict*) – The parent document. Should be a folder, user, or collection.
- **name** (*str*) – The name of the folder.
- **description** (*str*) – Description for the folder.
- **parentType** (*str*) – What type the parent is: ('folder' | 'user' | 'collection')
- **public** (*bool or None to inherit from parent*) – Public read access flag.
- **creator** (*dict*) – User document representing the creator of this folder.

- **allowRename** (*bool*) – if True and a folder or item of this name exists, automatically rename the folder.
- **reuseExisting** (*bool*) – If a folder with the given name already exists under the given parent, return that folder rather than creating a new one.

Returns The folder document that was created.

fileList(*doc*, *user=None*, *path=''*, *includeMetadata=False*, *subpath=True*, *mimeFilter=None*, *data=True*)

This function generates a list of 2-tuples whose first element is the relative path to the file from the folder's root and whose second element depends on the value of the *data* flag. If *data=True*, the second element will be a generator that will generate the bytes of the file data as stored in the assetstore. If *data=False*, the second element is the file document itself.

Parameters

- **doc** – The folder to list.
- **user** – The user used for access.
- **path** (*str*) – A path prefix to add to the results.
- **includeMetadata** (*bool*) – if True and there is any metadata, include a result which is the JSON string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the folder.
- **subpath** (*bool*) – if True, add the folder's name to the path.
- **mimeFilter** (*list or tuple*) – Optional list of MIME types to filter by. Set to None to include all files.
- **data** (*bool*) – If True return raw content of each file as stored in the assetstore, otherwise return file document.

Returns Iterable over files in this folder, where each element is a tuple of (path name of the file, stream function with file data or file object).

Return type generator(str, func)

getSizeRecursive(*folder*)

Calculate the total size of the folder by recursing into all of its descendant folders.

isOrphan(*folder*)

Returns True if this folder is orphaned (its parent is missing).

Parameters **folder** (*dict*) – The folder to check.

load(*id*, *level=2*, *user=None*, *objectId=True*, *force=False*, *fields=None*, *exc=False*)

We override load in order to ensure the folder has certain fields within it, and if not, we add them lazily at read time.

Parameters

- **id** (*string or ObjectId*) – The id of the resource.
- **user** (*dict or None*) – The user to check access against.
- **level** (AccessType) – The required access type for the object.
- **force** (*bool*) – If you explicitly want to circumvent access checking on this resource, set this to True.

move(*folder*, *parent*, *parentType*)

Move the given folder from its current parent to another parent object. Raises an exception if folder is an ancestor of parent.

Parameters

- **folder** (*dict*) – The folder to move.
- **parent** – The new parent object.
- **parentType** (*str*) – The type of the new parent object (user, collection, or folder).

parentsToRoot (*folder*, *curPath=None*, *user=None*, *force=False*, *level=0*)
Get the path to traverse to a root of the hierarchy.

Parameters **folder** (*dict*) – The folder whose root to find

Returns an ordered list of dictionaries from root to the current folder

remove (*folder*, *progress=None*, ***kwargs*)
Delete a folder recursively.

Parameters

- **folder** (*dict*) – The folder document to delete.
- **progress** (*girder.utility.progress.ProgressContext or None.*) – A progress context to record progress on.

setAccessList (*doc*, *access*, *save=False*, *recurse=False*, *user=None*, *progress=<girder.utility.progress.ProgressContext object>*, *setPublic=None*)
Overrides AccessControlledModel.setAccessList to add a recursive option. When *recurse=True*, this will set the access list on all subfolders to which the given user has ADMIN access level. Any subfolders that the given user does not have ADMIN access on will be skipped.

Parameters

- **doc** (folder) – The folder to set access settings on.
- **access** (*dict*) – The access control list.
- **save** (*bool*) – Whether the changes should be saved to the database.
- **recurse** (*bool*) – Whether this access list should be propagated to all subfolders underneath this folder.
- **user** – The current user (for recursive mode filtering).
- **progress** (*girder.utility.progress.ProgressContext*) – Progress context to update.
- **setPublic** (*bool or None*) – Pass this if you wish to set the public flag on the resources being updated.

setMetadata (*folder*, *metadata*)
Set metadata on a folder. A *ValidationException* is thrown in the cases where the metadata JSON object is badly formed, or if any of the metadata keys contains a period (‘.’).

Parameters

- **folder** (*dict*) – The folder to set the metadata on.
- **metadata** (*dict*) – A dictionary containing key-value pairs to add to the folder’s meta field

Returns the folder document

subtreeCount (*folder*, *includeItems=True*, *user=None*, *level=None*)
Return the size of the subtree rooted at the given folder. Includes the root folder in the count.

Parameters

- **folder** (*dict*) – The root of the subtree.
- **includeItems** (*bool*) – Whether to include items in the subtree count, or just folders.
- **user** – If filtering by permission, the user to filter against.
- **level** (*AccessLevel*) – If filtering by permission, the required permission level.

updateFolder(*folder*)
Updates a folder.

Parameters **folder** (*dict*) – The folder document to update

Returns The folder document that was edited.

updateSize(*doc*)
Recursively recomputes the size of this folder and its underlying folders and fixes the sizes as needed.

Parameters **doc** (*dict*) – The folder.

validate(*doc*, *allowRename=False*)
Validate the name and description of the folder, ensure that it is associated with a valid parent and that it has a unique name.

Parameters

- **doc** – the folder document to validate.
- **allowRename** – if True and a folder or item exists with the same name, rename the folder so that it is unique.

Returns the validated folder document.

Item

class girder.models.item.**Item**
Items are leaves in the data hierarchy. They can contain 0 or more files within them, and can also contain arbitrary metadata.

childFiles(*item*, *limit=0*, *offset=0*, *sort=None*, ***kwargs*)
Returns child files of the item. Passes any kwargs to the find function.

Parameters

- **item** – The parent item.
- **limit** – Result limit.
- **offset** – Result offset.
- **sort** – The sort structure to pass to pymongo.

copyItem(*srcItem*, *creator*, *name=None*, *folder=None*, *description=None*)
Copy an item, including duplicating files and metadata.

Parameters

- **srcItem** (*dict*) – the item to copy.
- **creator** – the user who will own the copied item.
- **name** (*str*) – The name of the new item. None to copy the original name.
- **folder** – The parent folder of the new item. None to store in the same folder as the original item.

- **description** (*str*) – Description for the new item. None to copy the original description.

Returns the new item.

createItem(*name*, *creator*, *folder*, *description=''*, *reuseExisting=False*)

Create a new item. The creator will be given admin access to it.

Parameters

- **name** (*str*) – The name of the item.
- **description** (*str*) – Description for the item.
- **folder** – The parent folder of the item.
- **creator** (*dict*) – User document representing the creator of the item.
- **reuseExisting** (*bool*) – If an item with the given name already exists under the given folder, return that item rather than creating a new one.

Returns The item document that was created.

fileList(*doc*, *user=None*, *path=''*, *includeMetadata=False*, *subpath=True*, *mimeFilter=None*, *data=True*)

This function generates a list of 2-tuples whose first element is the relative path to the file from the item's root and whose second element depends on the value of the *data* flag. If *data=True*, the second element will be a generator that will generate the bytes of the file data as stored in the assetstore. If *data=False*, the second element will be the file document itself.

Parameters

- **doc** – The item to list.
- **user** – A user used to validate data that is returned. This isn't used, but is present to be consistent across all model implementations of fileList.
- **path** (*str*) – A path prefix to add to the results.
- **includeMetadata** (*bool*) – If True and there is any metadata, include a result which is the JSON string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.
- **subpath** (*bool*) – If True and the item has more than one file, any metadata, or the sole file is not named the same as the item, then the returned paths include the item name.
- **mimeFilter** (*list or tuple*) – Optional list of MIME types to filter by. Set to None to include all files.
- **data** (*bool*) – If True return raw content of each file as stored in the assetstore, otherwise return file document.

Returns Iterable over files in this item, where each element is a tuple of (path name of the file, stream function with file data or file object).

Return type generator(str, func)

isOrphan(*item*)

Returns True if this item is orphaned (its folder is missing).

Parameters **item** (*dict*) – The item to check.

load(*id*, *level=2*, *user=None*, *objectId=True*, *force=False*, *fields=None*, *exc=False*)

Calls AccessControlMixin.load while doing some auto-correction.

Takes the same parameters as `girder.models.model_base.AccessControlledMixin.load()`.

move (*item*, *folder*)
Move the given item from its current folder into another folder.

Parameters

- **item** (*dict*) – The item to move.
- **folder** (*dict.*) – The folder to move the item into.

parentsToRoot (*item*, *user=None*, *force=False*)
Get the path to traverse to a root of the hierarchy.

Parameters

- **item** (*dict*) – The item whose root to find
- **user** (*dict or None*) – The user making the request (not required if force=True).
- **force** (*bool*) – Set to True to skip permission checking. If False, the returned models will be filtered.

Returns an ordered list of dictionaries from root to the current item

recalculateSize (*item*)
Recalculate the item size based on the files that are in it. If this is different than the recorded size, propagate the changes. :param item: The item to recalculate the size of. :returns: the recalculated size in bytes

remove (*item*, ***kwargs*)
Delete an item, and all references to it in the database.

Parameters **item** (*dict*) – The item document to delete.

setMetadata (*item*, *metadata*)
Set metadata on an item. A *ValidationException* is thrown in the cases where the metadata JSON object is badly formed, or if any of the metadata keys contains a period ('.').

Parameters

- **item** (*dict*) – The item to set the metadata on.
- **metadata** (*dict*) – A dictionary containing key-value pairs to add to the items meta field

Returns the item document

updateItem (*item*)
Updates an item.

Parameters **item** (*dict*) – The item document to update

Returns The item document that was edited.

updateSize (*doc*)
Recomputes the size of this item and its underlying files and fixes the sizes as needed.

Parameters **doc** (*dict*) – The item.

Setting

class girder.models.setting.**Setting**
This model represents server-wide configuration settings as key/value pairs.

get (*key*, *default='__default__'*)
Retrieve a setting by its key.

Parameters

- **key** (*str*) – The key identifying the setting.
- **default** – If no such setting exists, returns this value instead.

Returns The value, or the default value if the key is not found.

getDefault(*key*)
Retrieve the system default for a value.

Parameters **key** (*str*) – The key identifying the setting.

Returns The default value if the key is present in both SettingKey and referenced in SettingDefault; otherwise None.

reconnect()
Reconnect to the database and rebuild indices if necessary. If a unique index on key does not exist, make one, first discarding any extant index on key and removing duplicate keys if necessary.

set(*key*, *value*)
Save a setting. If a setting for this key already exists, this will replace the existing value.

Parameters

- **key** (*str*) – The key identifying the setting.
- **value** – The object to store for this setting.

Returns The document representing the saved Setting.

unset(*key*)
Remove the setting for this key. If no such setting exists, this is a no-op.

Parameters **key** (*str*) – The key identifying the setting to be removed.

validate(*doc*)
This method is in charge of validating that the setting key is a valid key, and that for that key, the provided value is valid. It first allows plugins to validate the setting, but if none of them can, it assumes it is a core setting and does the validation here.

static validateCorePluginsEnabled(*doc*)
Ensures that the set of plugins passed in is a list of valid plugin names. Removes any invalid plugin names, removes duplicates, and adds all transitive dependencies to the enabled list.

Assetstore

class girder.models.assetstore.**Assetstore**
This model represents an assetstore, an abstract repository of Files.

addComputedInfo(*assetstore*)
Add all runtime-computed properties about an assetstore to its document.

Parameters **assetstore** (*dict*) – The assetstore object.

getCurrent()
Returns the current assetstore. If none exists, this will raise a 500 exception.

importData(*assetstore*, *parent*, *parentType*, *params*, *progress*, *user*, ***kwargs*)
Calls the importData method of the underlying assetstore adapter.

list(*limit=0*, *offset=0*, *sort=None*)
List all assetstores.

Parameters

- **limit** – Result limit.
- **offset** – Result offset.
- **sort** – The sort structure to pass to pymongo.

Returns List of users.

remove(*assetstore*, ***kwargs*)

Delete an assetstore. If there are any files within this assetstore, a validation exception is raised.

Parameters **assetstore** (*dict*) – The assetstore document to delete.

File

class girder.models.file.**File**

This model represents a File, which is stored in an assetstore.

copyFile(*srcFile*, *creator*, *item=None*)

Copy a file so that we don't need to duplicate stored data.

Parameters

- **srcFile** (*dict*) – The file to copy.
- **creator** – The user copying the file.
- **item** – a new item to assign this file to (optional)

Returns a dict with the new file.

createFile(*creator*, *item*, *name*, *size*, *assetstore*, *mimeType=None*, *saveFile=True*, *reuseExisting=False*)

Create a new file record in the database.

Parameters

- **item** – The parent item.
- **creator** – The user creating the file.
- **assetstore** – The assetstore this file is stored in.
- **name** (*str*) – The filename.
- **size** (*int*) – The size of the file in bytes.
- **mimeType** (*str*) – The mimeType of the file.
- **saveFile** (*bool*) – if False, don't save the file, just return it.
- **reuseExisting** (*bool*) – If a file with the same name already exists in this location, return it rather than creating a new file.

createLinkFile(*name*, *parent*, *parentType*, *url*, *creator*)

Create a file that is a link to a URL, rather than something we maintain in an assetstore.

Parameters

- **name** (*str*) – The local name for the file.
- **parent** (*folder or item*) – The parent object for this file.
- **parentType** (*str*) – The parent type (folder or item)

- **url** – The URL that this file points to
- **creator** (*dict*) – The user creating the file.

download(*file*, *offset=0*, *headers=True*, *endByte=None*, *contentDisposition=None*, *extraParameters=None*)
Use the appropriate assetstore adapter for whatever assetstore the file is stored in, and call downloadFile on it. If the file is a link file rather than a file in an assetstore, we redirect to it.

Parameters

- **file** – The file to download.
- **offset** (*int*) – The start byte within the file.
- **headers** (*bool*) – Whether to set headers (i.e. is this an HTTP request for a single file, or something else).
- **endByte** (*int or None*) – Final byte to download. If None, downloads to the end of the file.
- **contentDisposition** (*str or None*) – Content-Disposition response header disposition-type value.

getAssetstoreAdapter(*file*)
Return the assetstore adapter for the given file.

isOrphan(*file*)
Returns True if this file is orphaned (its item or attached entity is missing).

Parameters **file** (*dict*) – The file to check.

propagateSizeChange(*item*, *sizeIncrement*, *updateItemSize=True*)
Propagates a file size change (or file creation) to the necessary parents in the hierarchy. Internally, this records subtree size in the item, the parent folder, and the root node under which the item lives. Should be called anytime a new file is added, a file is deleted, or a file size changes.

Parameters

- **item** (*dict*) – The parent item of the file.
- **sizeIncrement** (*int*) – The change in size to propagate.
- **updateItemSize** – Whether the item size should be updated. Set to False if you plan to delete the item immediately and don't care to update its size.

remove(*file*, *updateItemSize=True*, ***kwargs*)
Use the appropriate assetstore adapter for whatever assetstore the file is stored in, and call deleteFile on it, then delete the file record from the database.

Parameters

- **file** – The file document to remove.
- **updateItemSize** – Whether to update the item size. Only set this to False if you plan to delete the item and do not care about updating its size.

updateFile(*file*)
Call this when changing properties of an existing file, such as name or MIME type. This causes the updated stamp to change, and also alerts the underlying assetstore adapter that file information has changed.

updateSize(*file*)
Returns the size of this file. Does not currently check the underlying assetstore to verify the size.

Parameters **file** (*dict*) – The file.

Upload

class girder.models.upload.**Upload**

This model stores temporary records for uploads that have been approved but are not yet complete, so that they can be uploaded in chunks of arbitrary size. The chunks must be uploaded in order.

cancelUpload(*upload*)

Discard an upload that is in progress. This asks the assetstore to discard the data, then removes the item from the upload database.

Parameters **upload** (*dict*) – The upload document to remove.

createUpload(*user*, *name*, *parentType*, *parent*, *size*, *mimeType=None*, *reference=None*, *assetstore=None*, *attachParent=False*)

Creates a new upload record, and creates its temporary file that the chunks will be written into. Chunks should then be sent in order using the _id of the upload document generated by this method.

Parameters

- **user** (*dict*) – The user performing the upload.
- **name** (*str*) – The name of the file being uploaded.
- **parentType** (*str ('folder' or 'item')*) – The type of the parent being uploaded into.
- **parent** (*dict.*) – The document representing the parent.
- **size** (*int*) – Total size in bytes of the whole file.
- **mimeType** (*str*) – The mimeType of the file.
- **reference** (*str*) – An optional reference string that will be sent to the data.process event.
- **assetstore** – An optional assetstore to use to store the file. If unspecified, the current assetstore is used.
- **attachParent** (*boolean*) – if True, instead of creating an item within the parent or giving the file an itemId, set itemId to None and set attachedToType and attachedToId instead (using the values passed in parentType and parent). This is intended for files that shouldn't appear as direct children of the parent, but are still associated with it.

Returns The upload document that was created.

createUploadToFile(*file*, *user*, *size*, *reference=None*, *assetstore=None*)

Creates a new upload record into a file that already exists. This should be used when updating the contents of a file. Deletes any previous file content from the assetstore it was in. This will upload into the current assetstore rather than assetstore the file was previously contained in.

Parameters

- **file** – The file record to update.
- **user** – The user performing this upload.
- **size** – The size of the new file contents.
- **reference** (*str*) – An optional reference string that will be sent to the data.process event.
- **assetstore** – An optional assetstore to use to store the file. If unspecified, the current assetstore is used.

finalizeUpload(*upload*, *assetstore=None*)
This should only be called manually in the case of creating an empty file, i.e. one that has no chunks.

Parameters

- **upload** (*dict*) – The upload document.
- **assetstore** (*dict*) – If known, the containing assetstore for the upload.

Returns The file object that was created.

getTargetAssetstore(*modelType*, *resource*, *assetstore=None*)
Get the assetstore for a particular target resource, i.e. where new data within the resource should be stored. In Girder core, this is always just the current assetstore, but plugins may override this behavior to allow for more granular assetstore selection.

Parameters

- **modelType** – the type of the resource that will be stored.
- **resource** – the resource to be stored.
- **assetstore** – if specified, the prefered assetstore where the resource should be located. This may be overridden.

Returns the selected assetstore.

handleChunk(*upload*, *chunk*)
When a chunk is uploaded, this should be called to process the chunk. If this is the final chunk of the upload, this method will finalize the upload automatically.

Parameters

- **upload** (*dict*) – The upload document to update.
- **chunk** (*file*) – The file object representing the chunk that was uploaded.

list(*limit=0*, *offset=0*, *sort=None*, *filters=None*)
Search for uploads or simply list all visible uploads.

Parameters

- **limit** – Result set size limit.
- **offset** – Offset into the results.
- **sort** – The sort direction.
- **filters** – if not None, a dictionary that can contain ids that must match the uploads, plus an minimumAge value.

moveFileToAssetstore(*file*, *user*, *assetstore*)
Move a file from whatever assetstore it is located in to a different assetstore. This is done by downloading and re-uploading the file.

Parameters

- **file** – the file to move.
- **user** – the user that is authorizing the move.
- **assetstore** – the destination assetstore.

Returns the original file if it is not moved, or the newly 'uploaded' file if it is.

requestOffset(*upload*)
Requests the offset that should be used to resume uploading. This makes the request from the assetstore adapter.

untrackedUploads(*action='list'*, *assetstoreId=None*)
List or discard any uploads that an assetstore knows about but that our database doesn't have in it.

Parameters

- **action** (*str*) – 'delete' to discard the untracked uploads, anything else to just return with a list of them.
- **assetstoreId** (*str*) – if present, only include untracked items from the specified assetstore.

Returns a list of items that were removed or could be removed.

uploadFromFile(*obj*, *size*, *name*, *parentType=None*, *parent=None*, *user=None*, *mimeType=None*, *reference=None*, *assetstore=None*, *attachParent=False*)
This method wraps the entire upload process into a single function to facilitate "internal" uploads from a file-like object. Example:

```
size = os.path.getsize(filename)

with open(filename, 'rb') as f:
    self.model('upload').uploadFromFile(
        f, size, filename, 'item', parentItem, user)
```

Parameters

- **obj** (*file-like*) – The object representing the content to upload.
- **size** – The total size of
- **name** (*str*) – The name of the file to create.
- **parentType** (*str*) – The type of the parent: "folder" or "item".
- **parent** (*dict*) – The parent (item or folder) to upload into.
- **user** (*dict*) – The user who is creating the file.
- **mimeType** (*str*) – MIME type of the file.
- **reference** (*str*) – An optional reference string that will be sent to the data.process event.
- **assetstore** – An optional assetstore to use to store the file. If unspecified, the current assetstore is used.
- **attachParent** – if True, instead of creating an item within the parent or giving the file an itemId, set itemId to None and set attachedToType and attachedToId instead (using the values passed in parentType and parent). This is intended for files that shouldn't appear as direct children of the parent, but are still associated with it.

Events

This module contains the Girder events framework. It maintains a global mapping of events to listeners, and contains utilities for callers to handle or trigger events identified by a name.

Listeners should bind to events by calling:

```
girder.events.bind('event.name', 'my.handler', handlerFunction)
```

And events should be fired in one of two ways; if the event should be handled synchronously, fire it with:

```
girder.events.trigger('event.name', info)
```

And if the event should be handled asynchronously, use:

```
girder.events.daemon.trigger('event.name', info, callback)
```

For obvious reasons, the asynchronous method does not return a value to the caller. Instead, the caller may optionally pass the callback argument as a function to be called when the task is finished. That callback function will receive the Event object as its only argument.

class girder.events.**AsyncEventsThread**

This class is used to execute the pipeline for events asynchronously. This should not be invoked directly by callers; instead, they should use girder.events.daemon.trigger().

run()

Loops over all queued events. If the queue is empty, this thread gets put to sleep until someone calls trigger() on it with a new event to dispatch.

stop()

Gracefully stops this thread. Will finish the currently processing event before stopping.

trigger(*eventName*, *info=None*, *callback=None*)

Adds a new event on the queue to trigger asynchronously.

Parameters

- **eventName** – The event name to pass to the girder.events.trigger
- **info** – The info object to pass to girder.events.trigger

class girder.events.**Event**(*name*, *info*, *async=False*)

An Event object is created when an event is triggered. It is passed to each of the listeners of the event, which have a chance to add information to the event, and also optionally stop the event from being further propagated to other listeners, and also optionally instruct the caller that it should not execute its default behavior.

addResponse(*response*)

Listeners that wish to return data back to the caller who triggered this event should call this to append their own response to the event.

Parameters **response** – The response value, which can be any type.

preventDefault()

This can be used to instruct the triggerer of the event that the default behavior it would normally perform should not be performed. The semantics of this action are specific to the context of the event being handled, but a common use of this method is for a plugin to provide an alternate behavior that will replace the normal way the event is handled by the core system.

stopPropagation()

Listeners should call this on the event they were passed in order to stop any other listeners to the event from being executed.

girder.events.**bind**(*eventName*, *handlerName*, *handler*)

Bind a listener (handler) to the event identified by eventName. It is convention that plugins will use their own name as the handlerName, so that the trigger() caller can see which plugin(s) responded to the event.

Parameters

- **eventName** (*str*) – The name that identifies the event.
- **handlerName** (*str*) – The name that identifies the handler calling bind().

- **handler** (*function*) – The function that will be called when the event is fired. It must accept a single argument, which is the Event that was created by trigger(). This function should not return a value; any data that it needs to pass back to the triggerer should be passed via the addResponse() method of the Event.

girder.events.**bound**(**args*, ***kwds*)
A context manager to temporarily bind an event handler within its scope.

Parameters are the same as those to *girder.events.bind()*.

girder.events.**trigger**(*eventName*, *info=None*, *pre=None*, *async=False*)
Fire an event with the given name. All listeners bound on that name will be called until they are exhausted or one of the handlers calls the stopPropagation() method on the event.

Parameters

- **eventName** (*str*) – The name that identifies the event.
- **info** – The info argument to pass to the handler function. The type of this argument is opaque, and can be anything.
- **pre** (*function or None*) – A function that will be executed prior to the handler being executed. It will receive a dict with a "handler" key, (the function), "info" key (the info arg to this function), and "eventName" and "handlerName" values.
- **async** (*bool*) – Whether this event is executing on the background daemon (True) or on the request thread (False).

girder.events.**unbind**(*eventName*, *handlerName*)
Removes the binding between the event and the given listener.

Parameters

- **eventName** (*str*) – The name that identifies the event.
- **handlerName** (*str*) – The name that identifies the handler calling bind().

girder.events.**unbindAll**()
Clears the entire event map. All bound listeners will be unbound.

> **Warning:** This will also disable internal event listeners, which are necessary for normal Girder functionality. This function should generally never be called outside of testing.

Notification

class girder.models.notification.**Notification**
This model is used to represent a notification that should be streamed to a specific user in some way. Each notification contains a type field indicating what kind of notification it is, a userId field indicating which user the notification should be sent to, a data field representing the payload of the notification, a time field indicating the time at which the event happened, and an optional expires field indicating at what time the notification should be deleted from the database.

createNotification(*type*, *data*, *user*, *expires=None*, *token=None*)
Create a generic notification.

Parameters

- **type** (*str*) – The notification type.
- **data** – The notification payload.

- **user** (*dict*) – User to send the notification to.
- **expires** (*datetime.datetime*) – Expiration date (for transient notifications).
- **token** (*dict*) – Set this if the notification should correspond to a token instead of a user.

get(*user*, *since=None*, *token=None*)
Get outstanding notifications for the given user.

Parameters

- **user** – The user requesting updates. None to use the token instead.
- **since** (*datetime*) – Limit results to entities that have been updated since a certain timestamp.
- **token** – if the user is None, the token requesting updated.

initProgress(*user*, *title*, *total=0*, *state='active'*, *current=0*, *message=''*, *token=None*, *estimateTime=True*)
Create a "progress" type notification that can be updated anytime there is progress on some task. Progress records that are not updated for more than one hour will be deleted. The "time" field of a progress record indicates the time the task was started.

Parameters

- **user** – the user associated with this notification. If this is None, a session token must be specified.
- **title** (*str*) – The title of the task. This should not change over the course of the task. (e.g. 'Deleting folder "foo"')
- **total** (*int, long, or float*) – Some numeric value representing the total task length. By convention, setting this <= 0 means progress on this task is indeterminate.
- **state** (*ProgressState enum value.*) – Represents the state of the underlying task execution.
- **current** (*int, long, or float*) – Some numeric value representing the current progress of the task (relative to total).
- **message** (*str*) – Message corresponding to the current state of the task.
- **token** – if the user is None, associate this notification with the specified session token.
- **estimateTime** – if True, generate an estimate of the total time the task will take, if possible. If False, never generate a time estimate.

updateProgress(*record*, *save=True*, ***kwargs*)
Update an existing progress record.

Parameters

- **record** (*dict*) – The existing progress record to update.
- **total** (*int, long, or float*) – Some numeric value representing the total task length. By convention, setting this <= 0 means progress on this task is indeterminate. Generally this shouldn't change except in cases where progress on a task switches between indeterminate and determinate state.
- **state** (*ProgressState enum value.*) – Represents the state of the underlying task execution.
- **current** (*int, long, or float*) – Some numeric value representing the current progress of the task (relative to total).

- **increment** (*int, long, or float*) – Amount to increment the progress by. Don't pass both current and increment together, as that behavior is undefined.
- **message** (*str*) – Message corresponding to the current state of the task.
- **expires** (*datetime*) – Set a custom (UTC) expiration time on the record. Default is one hour from the current time.
- **save** (*bool*) – Whether to save the record to the database.

class girder.models.notification.**ProgressState**
Enum of possible progress states for progress records.

Python API for RESTful web API

Base Classes and Helpers

class girder.api.describe.**ApiDocs**(*templatePath=None*)
This serves up the Swagger page.

class girder.api.describe.**Description**(*summary*)
This class provides convenient chainable semantics to allow api route handlers to describe themselves to the documentation. A route handler function can apply the girder.api.describe.describeRoute decorator to itself (called with an instance of this class) in order to describe itself.

asDict()
Returns this description object as an appropriately formatted dict

errorResponse(*reason='A parameter was invalid.', code=400*)
This helper will build an errorResponse declaration for you. Many endpoints will be able to use the default parameter values for one of their responses.

Parameters

- **reason** (*str, list, or tuple*) – The reason or list of reasons why the error occurred.
- **code** (*int*) – HTTP status code.

pagingParams(*defaultSort, defaultSortDir=1, defaultLimit=50*)
Adds the limit, offset, sort, and sortdir parameter documentation to this route handler.

Parameters

- **defaultSort** (*str*) – The default field used to sort the result set.
- **defaultSortDir** (*int*) – Sort order: -1 or 1 (desc or asc)
- **defaultLimit** (*int*) – The default page size.

param(*name, description, paramType='query', dataType='string', required=True, enum=None, default=None*)
This helper will build a parameter declaration for you. It has the most common options as defaults, so you won't have to repeat yourself as much when declaring the APIs.

Note that we could expose more parameters from the Parameter Object spec, for example: format, allowEmptyValue, minimum, maximum, pattern, uniqueItems.

Parameters

- **name** – name of the parameter used in the REST query.
- **description** – explanation of the parameter.

- **paramType** – how is the parameter sent. One of 'query', 'path', 'body', 'header', or 'formData'.
- **dataType** – the data type expected in the parameter. This is one of 'integer', 'long', 'float', 'double', 'string', 'byte', 'binary', 'boolean', 'date', 'dateTime', 'password', or 'file'.
- **required** – True if the request will fail if this parameter is not present, False if the parameter is optional.
- **enum** – a fixed list of possible values for the field.

class girder.api.rest.**Resource**

All REST resources should inherit from this class, which provides utilities for adding resources/routes to the REST API.

boolParam(*key*, *params*, *default=None*)

Coerce a parameter value from a str to a bool. This function is case insensitive. The following string values will be interpreted as True:

- 'true'
- 'on'
- '1'
- 'yes'

All other strings will be interpreted as False. If the given param is not passed at all, returns the value specified by the default arg.

deleteAuthTokenCookie()

Helper method to kill the authentication cookie

ensureTokenScopes(*scope*)

Ensure that the token passed to this request is authorized for the designated scope or set of scopes. Raises an AccessException if not.

Parameters **scope** (*str or list of str*) – A scope or set of scopes that is required.

getBodyJson()

Bound wrapper for *girder.api.rest.getBodyJson()*.

getCurrentToken()

Returns the current valid token object that was passed via the token header or parameter, or None if no valid token was passed.

getCurrentUser(*returnToken=False*)

Returns the currently authenticated user based on the token header or parameter.

Parameters **returnToken** (*bool*) – Whether we should return a tuple that also contains the token.

Returns The user document from the database, or None if the user is not logged in or the token is invalid or expired. If returnToken=True, returns a tuple of (user, token).

getPagingParameters(*params*, *defaultSortField=None*, *defaultSortDir=1*)

Pass the URL parameters into this function if the request is for a list of resources that should be paginated. It will return a tuple of the form (limit, offset, sort) whose values should be passed directly into the model methods that are finding the resources. If the client did not pass the parameters, this always uses the same defaults of limit=50, offset=0, sort='name', sortdir=SortDir.ASCENDING=1.

Parameters

- **params** (*dict*) – The URL query parameters.
- **defaultSortField** (*str or None*) – If the client did not pass a 'sort' parameter, set this to choose a default sort field. If None, the results will be returned unsorted.
- **defaultSortDir** (*girder.constants.SortDir*) – Sort direction.

handleRoute (*method*, *path*, *params*)

Match the requested path to its corresponding route, and calls the handler for that route with the appropriate kwargs. If no route matches the path requested, throws a RestException.

This method fires two events for each request if a matching route is found. The names of these events are derived from the route matched by the request. As an example, if the user calls GET /api/v1/item/123, the following two events would be fired:

```
rest.get.item/:id.before
```

would be fired prior to calling the default API function, and

```
rest.get.item/:id.after
```

would be fired after the route handler returns. The query params are passed in the info of the before and after event handlers as event.info['params'], and the matched route tokens are passed in as dict items of event.info, so in the previous example event.info would also contain an 'id' key with the value of 123. For endpoints with empty sub-routes, the trailing slash is omitted from the event name, e.g.:

```
rest.post.group.before
```

Note: You will normally not need to call this method directly, as it is called by the internals of this class during the routing process.

Parameters

- **method** (*str*) – The HTTP method of the current request.
- **path** (list) – The path params of the request.

removeRoute (*method*, *route*, *handler=None*, *resource=None*)

Remove a route from the handler and documentation.

Parameters

- **method** (*str*) – The HTTP method, e.g. 'GET', 'POST', 'PUT'
- **route** (*tuple[str]*) – The route, as a list of path params relative to the resource root. Elements of this list starting with ':' are assumed to be wildcards.
- **handler** (*function*) – The method called for the route; this is necessary to remove the documentation.
- **resource** – the name of the resource at the root of this route.

requireAdmin (*user*, *message=None*)

Calling this on a user will ensure that they have admin rights. If not, raises an AccessException.

Parameters

- **user** (*dict.*) – The user to check admin flag on.
- **message** (*str or None*) – The exception message.

Raises **AccessException** – If the user is not an administrator.

requireParams(*required*, *provided*)
Throws an exception if any of the parameters in the required iterable is not found in the provided parameter set.

Parameters

- **required** (*list, tuple, or str*) – An iterable of required params, or if just one is required, you can simply pass it as a string.
- **provided** (*dict*) – The list of provided parameters.

route(*method*, *route*, *handler*, *nodoc=False*, *resource=None*)
Define a route for your REST resource.

Parameters

- **method** (*str*) – The HTTP method, e.g. 'GET', 'POST', 'PUT', 'PATCH'
- **route** (*tuple[str]*) – The route, as a list of path params relative to the resource root. Elements of this list starting with ':' are assumed to be wildcards.
- **handler** (*function*) – The method to be called if the route and method are matched by a request. Wildcards in the route will be expanded and passed as kwargs with the same name as the wildcard identifier.
- **nodoc** (*bool*) – If your route intentionally provides no documentation, set this to True to disable the warning on startup.
- **resource** – The name of the resource at the root of this route.

sendAuthTokenCookie(*user*, *scope=None*)
Helper method to send the authentication cookie

setRawResponse(**args*, ***kwargs*)
Bound alias for `girder.api.rest.setRawResponse`.

exception `girder.api.rest.`**RestException**(*message*, *code=400*, *extra=None*)
Throw a RestException in the case of any sort of incorrect request (i.e. user/client error). Login and permission failures should set a 403 code; almost all other validation errors should use status 400, which is the default.

class `girder.api.rest.`**boundHandler**(*ctx=None*)
This decorator allows unbound functions to be conveniently added as route handlers to existing *girder.api.rest.Resource* instances. With no arguments, this uses a shared, generic `Resource` instance as the context. If you need a specific instance, pass that as the `ctx` arg, for instance if you need to reference the resource name or any other properties specific to a Resource subclass.

Plugins that add new routes to existing API resources are encouraged to use this to gain access to bound convenience methods like `self.model`, `self.boolParam`, `self.requireParams`, etc.

`girder.api.rest.`**endpoint**(*fun*)
REST HTTP method endpoints should use this decorator. It converts the return value of the underlying method to the appropriate output format and sets the relevant response headers. It also handles RestExceptions, which are 400-level exceptions in the REST endpoints, AccessExceptions resulting from access denial, and also handles any unexpected errors using 500 status and including a useful traceback in those cases.

If you want a streamed response, simply return a generator function from the inner method.

`girder.api.rest.`**ensureTokenScopes**(*token*, *scope*)
Call this to validate a token scope for endpoints that require tokens other than a user authentication token. Raises an AccessException if the required scopes are not allowed by the given token.

Parameters

- **token** (*dict*) – The token object used in the request.

- **scope** (*str or list of str*) – The required scope or set of scopes.

girder.api.rest.**getApiUrl**(*url=None*)

In a request thread, call this to get the path to the root of the REST API. The returned path does *not* end in a forward slash.

Parameters **url** – URL from which to extract the base URL. If not specified, uses *cherrypy.url()*

girder.api.rest.**getBodyJson**(*allowConstants=False*)

For requests that are expected to contain a JSON body, this returns the parsed value, or raises a *girder.api.rest.RestException* for invalid JSON.

Parameters **allowConstants** (*bool*) – Whether the keywords Infinity, -Infinity, and NaN should be allowed. These keywords are valid JavaScript and will parse to the correct float values, but are not valid in strict JSON.

girder.api.rest.**getUrlParts**(*url=None*)

Calls urllib.parse.urlparse on a URL.

Parameters **url** (*str or None*) – A URL, or None to use the current request's URL.

Returns The URL's separate components.

Return type urllib.parse.ParseResult

Note: This is compatible with both Python 2 and 3.

---

girder.api.rest.**iterBody**(*length=65536*, *strictLength=False*)

This is a generator that will read the request body a chunk at a time and yield each chunk, abstracting details of the underlying HTTP server. This function works regardless of whether the body was sent with a Content-Length or using Transfer-Encoding: chunked, but the behavior is slightly different in each case.

If *Content-Length* is provided, the *length* parameter is used to read the body in chunks up to size *length*. This will block until end of stream or the specified number of bytes is ready.

If *Transfer-Encoding: chunked* is used, the *length* parameter is ignored by default, and the generator yields each chunk that is sent in the request regardless of its length. However, if *strictLength* is set to True, it will block until *length* bytes have been read or the end of the request.

Parameters

- **length** (*int*) – Max buffer size to read per iteration if the request has a known *Content-Length*.
- **strictLength** (*bool*) – If the request is chunked, set this to True to block until `length` bytes have been read or end-of-stream.

class girder.api.rest.**loadmodel**(*map=None*, *model=None*, *plugin='_core'*, *level=None*, *force=False*, *exc=True*, ***kwargs*)

This is a decorator that can be used to load a model based on an ID param. For access controlled models, it will check authorization for the current user. The underlying function is called with a modified set of keyword arguments that is transformed by the "map" parameter of this decorator. Any additional kwargs will be passed to the underlying model's *load*.

Parameters

- **map** (*dict or None*) – Map of incoming parameter name to corresponding model arg name. If None is passed, this will map the parameter named "id" to a kwarg named the same as the "model" parameter.
- **model** (*str*) – The model name, e.g. 'folder'

- **plugin** (*str*) – Plugin name, if loading a plugin model.
- **level** (AccessType) – Access level, if this is an access controlled model.
- **force** (*bool*) – Force loading of the model (skip access check).
- **exc** (*bool*) – Whether an exception should be raised for a nonexistent resource.

girder.api.rest.**rawResponse**(*fun*)
This is a decorator that can be placed on REST route handlers, and is equivalent to calling setRawResponse() in the handler body.

girder.api.rest.**requireAdmin**(*user*, *message=None*)
Calling this on a user will ensure that they have admin rights. If not, raises an AccessException.

Parameters

- **user** (*dict.*) – The user to check admin flag on.
- **message** (*str or None*) – The exception message.

Raises **AccessException** – If the user is not an administrator.

girder.api.rest.**setRawResponse**(*val=True*)
Normally, non-streaming responses go through a serialization process in accordance with the "Accept" request header. Endpoints that wish to return a raw response without using a streaming response should call this, or use its bound version on the Resource class, or add the rawResponse decorator on the REST route handler function.

Parameters **val** (*bool*) – Whether the return value should be sent raw.

girder.api.rest.**setResponseHeader**(*header*, *value*)
Set a response header to the given value.

Parameters

- **header** (*str*) – The header name.
- **value** (*str*) – The value for the header.

User

class girder.api.v1.user.**User**
API Endpoint for users in the system.

login(*params*)
Login endpoint. Sends an auth cookie in the response on success. The caller is expected to use HTTP Basic Authentication when calling this endpoint.

Group

class girder.api.v1.group.**Group**
API Endpoint for groups.

find(**args*, ***kwargs*)
List or search for groups.

Parameters **params** (*dict*) – Request query parameters.

Returns A page of matching Group documents.

Item

Folder

class girder.api.v1.folder.**Folder**
API Endpoint for folders.

createFolder(**iargs*, ***ikwargs*)
Create a new folder.

Parameters

- **parentId** (*str*) – The _id of the parent folder.
- **parentType** (*str - 'user', 'collection', or 'folder'*) – The type of the parent of this folder.
- **name** – The name of the folder to create.
- **description** – Folder description.
- **public** (*bool*) – Public read access flag.

downloadFolder(**args*, ***kwargs*)
Returns a generator function that will be used to stream out a zip file containing this folder's contents, filtered by permissions.

find(**args*, ***kwargs*)
Get a list of folders with given search parameters. Currently accepted search modes are:

1. Searching by parentId and parentType, with optional additional filtering by the name field (exact match) or using full text search within a single parent folder. Pass a "name" parameter or "text" parameter to invoke these additional filters.
2. Searching with full text search across all folders in the system. Simply pass a "text" parameter for this mode.

Utility

class girder.utility.model_importer.**ModelImporter**
Any class that wants to have convenient model importing semantics should extend/mixin this class.

static model(*model*, *plugin=None*)
Call this to get the instance of the specified model. It will be lazy-instantiated.

Parameters

- **model** (*string*) – The name of the model to get. This is the module name, e.g. "folder". The class name must be the upper-camelcased version of that module name, e.g. "Folder".
- **plugin** – If the model you wish to load is a model within a plugin, set this to the name of the plugin containing the model.

Returns The instantiated model, which is a singleton.

static registerModel(*model*, *instance*, *plugin='_core'*)
Use this method to manually register a model singleton instead of having it automatically discovered.

Parameters

- **model** (*str*) – The model name.
- **plugin** (*str*) – If a plugin model, pass the canonical plugin name.

- **instance** (*subclass of Model*) – The model singleton instance.

girder.utility.model_importer.**reinitializeAll**()
Force all models to reconnect/rebuild indices (needed for testing).

girder.utility.server.**configureServer**(*test=False*, *plugins=None*, *curConfig=None*)
Function to setup the cherrypy server. It configures it, but does not actually start it.

Parameters

- **test** (*bool*) – Set to True when running in the tests.
- **plugins** – If you wish to start the server with a custom set of plugins, pass this as a list of plugins to load. Otherwise, will use the PLUGINS_ENABLED setting value from the db.
- **curConfig** – The configuration dictionary to update.

girder.utility.server.**loadRouteTable**(*reconcileRoutes=False*)
Retrieves the route table from Girder and reconciles the state of it with the current application state.

Reconciliation deals with 2 scenarios: 1) A plugin is no longer active (by being disabled or removed) and the route for the plugin needs to be removed. 2) A webroot was added (a new plugin was enabled) and a default route needs to be added.

Returns The non empty routes (as a dict of name -> route) to be mounted by CherryPy

during Girder's setup phase.

girder.utility.server.**setup**(*test=False*, *plugins=None*, *curConfig=None*)
Configure and mount the Girder server and plugins under the appropriate routes.

See ROUTE_TABLE setting.

Parameters

- **test** – Whether to start in test mode.
- **plugins** – List of plugins to enable.
- **curConfig** – The config object to update.

girder.utility.server.**staticFile**(*path*, *contentType=None*)
Helper function to serve a static file. This should be bound as the route object, i.e. info['serverRoot'].route_name = staticFile('...')

Parameters

- **path** (*str*) – The path of the static file to serve from this route.
- **contentType** – The MIME type of the static file. If set to None, the content type wll be guessed by the file extension of the 'path' argument.

girder.utility.mail_utils.**addTemplateDirectory**(*dir*, *prepend=False*)
Adds a directory to the search path for mail templates. This is useful for plugins that have their own set of mail templates.

Parameters

- **dir** (*str*) – The directory to add to the template lookup path.
- **prepend** (*bool*) – If True, adds this directory at the beginning of the path so that it will override any existing templates with the same name. Otherwise appends to the end of the lookup path.

girder.utility.mail_utils.**getEmailUrlPrefix**()
Return the URL prefix for links back to the server. This is the link to the server root, so Girder-level path information and any query parameters or fragment value should be appended to this value.

girder.utility.mail_utils.**renderTemplate**(*name*, *params=None*)
Renders one of the HTML mail templates located in girder/mail_templates.

Parameters

- **name** – The name of the file inside girder/mail_templates to render.
- **params** (*dict*) – The parameters to pass when rendering the template.

Returns The rendered template as a string of HTML.

girder.utility.mail_utils.**sendEmail**(*to=None*, *subject=None*, *text=None*, *toAdmins=False*, *bcc=None*)
Send an email. This builds the appropriate email object and then triggers an asynchronous event to send the email (handled in _sendmail).

Parameters

- **to** (*str, list/tuple, or None*) – The recipient's email address, or a list of addresses.
- **subject** (*str*) – The subject line of the email.
- **text** (*str*) – The body of the email.
- **toAdmins** (*bool*) – To send an email to all site administrators, set this to True, which will override any "to" argument that was passed.
- **bcc** (*str, list/tuple, or None*) – Recipient email address(es) that should be specified using the Bcc header.

class girder.utility.progress.**ProgressContext**(*on*, *interval=0.5*, ***kwargs*)
This class is a context manager that can be used to update progress in a way that rate-limits writes to the database and guarantees a flush when the context is exited. This is a no-op if "on" is set to False, which is meant as a convenience for callers. Any additional kwargs passed to this constructor are passed through to the `initProgress` method of the notification model.

Parameters

- **on** (*bool*) – Whether to record progress.
- **interval** (*int or float*) – Minimum time interval at which to write updates to the database, in seconds.
- **user** (*dict*) – The user creating this progress.
- **title** (*str*) – The title for the task being tracked.

update(*force=False*, ***kwargs*)
Update the underlying progress record. This will only actually save to the database if at least self.interval seconds have passed since the last time the record was written to the database. Accepts the same kwargs as Notification.updateProgress.

Parameters **force** (*bool*) – Whether we should force the write to the database. Use only in cases where progress may be indeterminate for a long time.

girder.utility.progress.**setResponseTimeLimit**(*duration=600*, *onlyExtend=True*)
If we are currently within a cherrypy response, extend the time limit. By default, cherrypy responses will timeout after 300 seconds, so any activity which can take longer should call this function.

Note that for cherrypy responses that include streaming generator functions, such as downloads, the timeout is only relevant until the first `yield` is reached. As such, long running generator responses do not generally need to call this function.

Parameters

- **duration** – additional duration in seconds to allow for the response.
- **onlyExtend** – if True, only ever increase the timeout. If False, the new duration always replaces the old one.

This module contains utility methods for parsing girder path strings.

exception girder.utility.path.**NotFoundException**(*message*, *field=None*)
A special case of ValidationException representing the case when the resource at a given path does not exist.

girder.utility.path.**decode**(*token*)
Unescape special characters in a token from a path representation.

Parameters **token** (*str*) – The token to decode

Returns The decoded string

Return type str

girder.utility.path.**encode**(*token*)
Escape special characters in a token for path representation.

Parameters **token** (*str*) – The token to encode

Returns The encoded string

Return type str

girder.utility.path.**join**(*tokens*)
Join a list of tokens into an encoded path string.

Parameters **tokens** (list) – A list of tokens

Returns The encoded path string

Return type str

girder.utility.path.**lookUpPath**(*path*, *user=None*, *test=False*, *filter=True*)
Look up a resource in the data hierarchy by path.

Parameters

- **path** – path of the resource
- **user** – user with correct privileges to access path
- **test** (*bool*) – defaults to false, when set to true will return None instead of throwing exception when path doesn't exist
- **filter** (*bool*) – Whether the returned model should be filtered.

girder.utility.path.**lookUpToken**(*token*, *parentType*, *parent*)
Find a particular child resource by name or throw an exception.

Parameters

- **token** – the name of the child resource to find
- **parentType** – the type of the parent to search
- **parent** – the parent resource

Returns the child resource

girder.utility.path.**split**(*path*)
Split an encoded path string into decoded tokens.

Parameters **path** (*str*) – An encoded path string

Returns A list of decoded tokens

Return type *list*

Constants

Constants should be defined here.

class girder.constants.**AccessType**
Represents the level of access granted to a user or group on an AccessControlledModel. Having a higher access level on a resource also confers all of the privileges of the lower levels.

Semantically, READ access on a resource means that the user can see all the information pertaining to the resource, but cannot modify it.

WRITE access usually means the user can modify aspects of the resource.

ADMIN access confers total control; the user can delete the resource and also manage permissions for other users on it.

class girder.constants.**AssetstoreType**
All possible assetstore implementation types.

class girder.constants.**CoreEventHandler**
This enum represents handler identifier strings for core event handlers. If you wish to unbind a core event handler, use one of these as the `handlerName` argument. Unbinding core event handlers can be used to disable certain default functionalities.

girder.constants.**STATIC_ROOT_DIR = '/home/docs/checkouts/readthedocs.org/user_builds/girder/checkouts/v2.0'**
The local directory containing the static content. Should contain `clients/web/static`.

class girder.constants.**SettingDefault**
Core settings that have a default should be enumerated here with the SettingKey.

class girder.constants.**SettingKey**
Core settings should be enumerated here by a set of constants corresponding to sensible strings.

class girder.constants.**TerminalColor**
Provides a set of values that can be used to color text in the terminal.

class girder.constants.**TokenScope**
Constants for core token scope strings. Token scopes must not contain spaces, since many services accept scope lists as a space-separated list of strings.

classmethod describeScope(*scopeId*, *name*, *description*, *admin=False*)
Register a description of a scope.

Parameters

- **scopeId** (*str*) – The unique identifier string for the scope.
- **name** (*str*) – A short human readable name for the scope.
- **description** (*str*) – A more complete description of the scope.
- **admin** (*bool*) – If this scope only applies to admin users, set to True.

Clients

jQuery Plugins

There are a set of jQuery plugins that interact with the Girder API. These can be found in the `clients/jquery` directory of the source tree.

`$.girderBrowser(cfg)`

Arguments

- **cfg** (*object*) – Configuration object
- **caret** (*boolean*) – Draw a caret on main menu to indicate dropdown (*true* by default).
- **label** (*string*) – The text to display in the main menu dropdown.
- **api** (*string*) – The root path to the Girder API (*/api/v1* by default).
- **selectItem** (*function(item, api)*) – A function to call when an item is clicked. It will be passed the item's information and the API root.
- **selectFolder** (*function(folder, api)*) – A function to call when a folder is clicked. It will be passed the folder's information and the API root.
- **search** (*boolean*) – Include a search box for gathering general string search results.
- **selectSearchResult** (*function(result, api)*) – A function to call when a search result is clicked. It will be passed the result item's information and the API root.

This plugin creates a Bootstrap dropdown menu reflecting the current contents of a Girder server as accessible by the logged-in user. The selection on which this plugin is invoked should be an `<li>` element that is part of a Bootstrap navbar. For example:

```
<div class="navbar navbar-default navbar-fixed-top">
    <div class=navbar-header>
        <a class=navbar-brand href=/examples>Girder</a>
    </div>

    <ul class="nav navbar-nav">
        <li id=girder-browser>
            <a>Dummy</a>
        </li>
    </ul>
</div>
```

Then, in a JavaScript file:

```
$("#girder-browser").girderBrowser({
    // Config options here
    //     .
    //     .
    //     .
});
```

The anchor text "dummy" in the example HTML will appear in the rendered page if the plugin fails to execute for any reason. This is purely a debugging measure - since the plugin empties the target element before it creates the menu, the anchor tag (or any other content) is not required.

3.3.2 Developer Guide

Girder is a platform-centric web application whose client and server are very loosely coupled. As such, development of Girder can be divided into the server (a CherryPy-based Python module) and the primary client (a Backbone-based) web client. This section is intended to get prospective contributors to understand the tools used to develop Girder.

Configuring Your Development Environment

In order to develop Girder, you should first refer to the System Prerequisites and Installation sections to setup a basic local environment.

Next, you should install the Python development dependencies, to provide helpful development tools and to allow the test suite to run:

```
pip install -r requirements-dev.txt
```

During development, once Girder is started via `python -m girder`, the server will reload itself whenever a Python file is modified.

Girder's web-based client application is built using the Grunt task running tool. When you run the `npm install` command during Girder's installation, it will run all of the grunt tasks required to build the web client. Grunt tasks are run with the `grunt` executable, which is installed under your Girder source directory in the `./node_modules/.bin/` directory. You could conveniently update the `PATH` by running `export PATH=$(pwd)/node_modules/.bin:$PATH` – once you do that, you can just type `grunt` in your shell to run tasks.

Note: Alternatively, you could install the grunt command line interface globally so that the `grunt` command is automatically added to your `PATH`. If you want to do that, run `npm install -g grunt-cli`. Note that this command requires `sudo` on many systems.

It is recommended during development to make use of the `npm run watch` tool. Running `grunt watch` in the root of the repository will watch for JavaScript, Stylus, and Pug changes in order to rebuild them on-the-fly. If you do not run `npm run watch` while making code changes, you will need to run the `npm run build` command to manually rebuild the web client in order to see your changes reflected.

Vagrant

A shortcut to going through the installation steps for development is to use Vagrant to setup the environment on a VirtualBox virtual machine. To setup this environment run `vagrant up` in the root of the repository. This will spin up and provision a virtual machine, provided you have Vagrant and VirtualBox installed. Once this process is complete, you can run `vagrant ssh` in order to start Girder. There is a helper script in the Vagrant home directory that will start Girder in a detached screen session. You may want to run a similar process to run `grunt watch` as detailed above.

Utilities

Girder has a set of utility modules and classes that provide handy extractions for certain functionality. Detailed API documentation can be found *here*.

Configuration Loading

The Girder configuration loader allows for lazy-loading of configuration values in a CherryPy-agnostic manner. The recommended idiom for getting the config object is:

```
from girder.utility import config
cur_config = config.getConfig()
```

There is a configuration file for Girder located in **girder/conf**. The file **girder.dist.cfg** is the file distributed with the repository and containing the default configuration values. This file should not be edited when deploying Girder. Rather, edit the **girder.local.cfg** file. You only need to edit the values in the file that you wish to change from their default values; the system loads the **dist** file first, then the **local** file, so your local settings will override the defaults.

Server Development

All commits to the core python code must work in both python 2.7 and 3.4. Python code in plugins should also work in both, but some plugins may depend on third party libraries that do not support python 3. If that is the case, those plugins should declare `"python3": false` in their **plugin.json** or **plugin.yml** file to indicate that they do not support being run in python 3. Automated testing of those plugins should also be disabled for python3 if those tests would fail in a python 3 environment. This can be achieved by passing an additional flag `PY2_ONLY` to `add_python_test` in your **plugin.cmake** file.

Python Style

We use `flake8` to test for Python style on the server side.

Use `%` instead of `format`

Use `%` or some other string formatting operation that coerces to unicode, and avoid `format`, since it does not coerce to unicode and has caused bugs.

Client Development

If you are writing a custom client application that communicates with the Girder REST API, you should look at the Swagger page that describes all of the available API endpoints. The Swagger page can be accessed by navigating a web browser to `api/v1` relative to the server root. If you wish to consume the Swagger-compliant API specification programmatically, the JSON listing is served out of `api/v1/describe`.

If you are working on the main Girder web client, either in core or extending it via plugins, there are a few conventions that should be followed. Namely, if you write code that instantiates new `girder.View` descendant objects, you should pass a `parentView` property when constructing it. This will allow the child view to be cleaned up recursively when the parent view is destroyed. If you forget to set the `parentView` property when constructing the view, the view will still work as expected, but a warning message will appear in the console to remind you. Example:

```
MySubView = girder.View.extend({
    ...
});

new MySubView({
    el: ...,
    otherProperty: ...,
    parentView: this
});
```

If you use `girder.View` in custom Backbone apps and need to create a new root view object, set the `parentView` to `null`. If you are using a Girder widget in a custom app that does not use the `girder.View` as the base object for its views, you should pass `parentView: null` and make sure to call `destroy()` on the view manually when it should be cleaned up.

Server Side Testing

Running the Tests

First, you will need to configure the project with CMake.

```
mkdir ../girder-build
cd ../girder-build
cmake ../girder
```

You only need to do this once. From then on, whenever you want to run the tests, just:

```
cd girder-build
ctest
```

There are many ways to filter tests when running CTest, or run the tests in parallel. More information about CTest can be found here.

If you run into errors on any of the packaging tests, two possible fixes are

1) run `make` inside your `girder-build` directory, which will create a special virtualenv needed to build the packages.

2) delete any of the files generated by the packaging tests, which will be in your source dir `girder` and could include `girder-<version>.tar.gz`, `girder-web-<version>.tar.gz`, and `girder-plugins-<version>.tar.gz`.

Running the Tests with Coverage Tracing

To run Python coverage on your tests, configure with CMake and run CTest. The coverage data will be automatically generated. After the tests are run, you can find the HTML output from the coverage tool in the source directory under **/clients/web/dev/built/py_coverage**.

Client Side Testing

Using the same setup as above for the Server Side Tests, your environment will be set up to run client side tests. Running

```
cd girder-build
ctest
```

will run all of the tests, which include the client side tests. Our client tests use the Jasmine JS testing framework.

When running client side tests, if you try to SIGINT (ctrl+c) the CTest process, CTest won't pass that signal down to the test processes for them to handle. This can result in orphaned python unittest processes and can prevent future runs of client tests. If you run a client side test and see an error message similar to `IOError: Port 30015 not free on '0.0.0.0'`, then look for an existing process similar to `/usr/bin/python2.7 -m unittest -v tests.web_client_test`, kill the process, and then try your tests again.

Adding a New Client Side Test

To add a new client side test, add a new spec file in `/clients/web/test/spec/`, add a line referencing your spec file to `/girder/tests/CMakeLists.txt` using the `add_web_client_test` function, and then run in your build directory

```
cmake ../girder
```

before running your tests.

An example of a very simple client side test would be as follows

```
add_web_client_test(some_client_test "someSpec.js" PLUGIN "my_plugin")
```

The `PLUGIN` argument indicates that "my_plugin" is the owner of `some_client_test`, at the time of the test my_plugin and all of its dependencies will be loaded.

If additional plugins are needed for a specific test, that can be achieved using the `ENABLEDPLUGINS` argument

```
add_web_client_test(another_client_test "anotherSpec.js" PLUGIN "my_plugin" ENABLEDPLUGINS "my_plugin
```

Here `ENABLEDPLUGINS` ensures that my_plugin *and* the jobs plugin are loaded, along with their dependencies at the time of `another_client_test`.

Note: Core functionality shouldn't depend on plugins being enabled, this test definition is more suitable for a plugin. Information for testing plugins can be found under Plugin Development.

You will find many useful methods for client side testing in the `girderTest` object defined at `/clients/web/test/testUtils.js`.

Code Review

Contributions to Girder are done via pull requests with a core developer accepting a PR by saying it "Looks good to me" or LGTM. At this point, the topic branch can be merged to master. This is meant to be a simple, low-friction process; however, code review is very important. It should be done carefully and not taken lightly. Thorough code review is a crucial part of developing quality software. When performing a code review, ask the following:

1. Is the continuous integration server happy with this?
2. Are there tests for this feature or bug fix?
3. Is this documented (for users and/or developers)?
4. Are the commits modular with good notes?
5. Will this merge cleanly?
6. Does this break backward compatibility? Is that okay?
7. What are the security implications of this change? Does this open Girder up to any vulnerabilities (XSS, CSRF, DB Injection, etc)?

Third-Party Libraries

Girder's standard procedure is to use a tool like piprot to check for out-of-date third-party library requirements on a quarterly basis (typically near the dates of the solstices and equinoxes). Library packages should generally be upgraded to the latest released version, except when:

- Doing so would introduce any new unfixable bugs or regressions.
- Other closely-affiliated projects (e.g. Romanesco, Minerva) use the same library *and* the other project cannot also feasibly be upgraded simultaneously.
- The library has undergone a major API change, and development resources do not permit updating Girder accordingly *or* Girder exposes parts of the library as members of Girder's API surface (e.g. CherryPy) and upgrading would cause incompatible API changes to be exposed. In this case, the library should still be upgraded to the highest non-breaking version that is available at the time.

Note: In the event that a security vulnerability is discovered in a third-party library used by Girder, the library *must* be upgraded to patch the vulnerability immediately and without regard to the aforementioned exceptions. However, attempts should still be made to maintain API compatibility via monkey patching, wrapper classes, etc.

Creating a new release

Girder releases are uploaded to PyPI for easy installation via `pip`. In addition, the python source packages are stored as releases inside the official github repository. The recommended process for generating a new release is described here.

1. From the target commit, set the desired version number in `package.json` and `girder/__init__.py`. Create a new commit and note the SHA; this will become the release tag.
2. Ensure that all tests pass.
3. Clone the repository in a new directory and checkout the release SHA. (Packaging in an old directory could cause files and plugins to be mistakenly included.)
4. Run `npm install && grunt package`. This will generate the source distribution tarball with a name like `girder-<version>.tar.gz`.
5. Create a new virtual environment and install the python package into it and build the web client. This should not be done in the repository directory because the wrong Girder package will be imported.

```
mkdir test && cd test
virtualenv release
source release/bin/activate
pip install ../girder-<version>.tar.gz
girder-install web
```

6. Now start up the Girder server and ensure that you can browse the web client, plugins, and swagger docs.
7. When you are confident everything is working correctly, generate a new release on GitHub. You must be sure to use a tag version of `v<version>`, where `<version>` is the version number as it exists in `package.json`. For example, `v0.2.4`. Attach the tarball you generated to the release.
8. Add the tagged version to readthedocs and make sure it builds correctly.
9. Finally, upload the release to PyPI with the following command:

```
python setup.py sdist upload
```

Releasing the python client package

Whenever the main Girder package is released, the python client package should also be versioned and released if is has changed since the last Girder release or the last time it was released. Normal semantic versioning is not in use for

the python client package because its version is partially dependent on the Girder server package version. The rules for versioning the python client package are as follows:

- The major version of the python client should be the same as the major version of the Girder server package, assuming it is compatible with the server API.
- The minor version should be incremented if there is any change in backward compatibility within the python client API, or if significant new features are added.
- If the release only includes bug fixes or minor enhancements, just increment the patch version token.

The process for releasing the python client is as follows:

1. Set the version number inside `clients/python/girder_client/__init__.py` according to the above rules. It is set in the line near the top of the file that looks like `__version__ = 'x.y.z'`
2. Change to the `clients/python` directory of the source tree and build the package using the following commands.

```
cd clients/python
python setup.py sdist --dist-dir .
```

3. That should have created the package tarball as `girder-client-<version>.tar.gz`. Install it locally in a virtualenv and ensure that you can call the `girder-cli` executable.

```
mkdir test && cd test
virtualenv release
source release/bin/activate
pip install ../girder-client-<version>.tar.gz
girder-cli
```

4. Go back to the `clients/python` directory and upload the package to pypi:

```
cd ..
python setup.py sdist upload
```

3.3.3 Plugin Development

The capabilities of Girder can be extended via plugins. The plugin framework is designed to allow Girder to be as flexible as possible, on both the client and server sides.

A plugin is self-contained in a single directory. To create your plugin, simply create a directory within the **plugins** directory. In fact, that directory is the only thing that is truly required to make a plugin in Girder. All of the other components discussed henceforth are optional.

Example Plugin

We'll use a contrived example to demonstrate the capabilities and components of a plugin. Our plugin will be called *cats*.

```
cd plugins ; mkdir cats
```

The first thing we should do is create a plugin config file in the **cats** directory. As promised above, this file is not required, but is strongly recommended by convention. This file contains high-level information about your plugin, and can be either JSON or YAML. If you want to use YAML features, make sure to name your config file `plugin.yml` instead of `plugin.json`. For our example, we'll just use JSON.

```
touch cats/plugin.json
```

The plugin config file should specify a human-readable name and description for your plugin, and can optionally contain a list of other plugins that your plugin depends on. If your plugin has dependencies, the other plugins will be enabled whenever your plugin is enabled. The contents of plugin.json for our example will be:

Note: If you have both `plugin.json` and `plugin.yml` files in the directory, the `plugin.json` will take precedence.

```
{
"name": "My Cats Plugin",
"description": "Allows users to manage their cats.",
"version": "1.0.0",
"dependencies": ["other_plugin"]
}
```

This information will appear in the web client administration console, and administrators will be able to enable and disable it there. Whenever plugins are enabled or disabled, a server restart is required in order for the change to take effect.

Extending the Server-Side Application

Girder plugins can augment and alter the core functionality of the system in almost any way imaginable. These changes can be achieved via several mechanisms which are described below. First, in order to implement the functionality of your plugin, create a **server** directory within your plugin, and make it a Python package by creating **__init__.py**.

```
cd cats ; mkdir server ; touch server/__init__.py
```

This package will be imported at server startup if your plugin is enabled. Additionally, if your package implements a `load` function, that will be called. This `load` function is where the logic of extension should be performed for your plugin.

```
def load(info):
    ...
```

This `load` function must take a single argument, which is a dictionary of useful information passed from the core system. This dictionary contains an `apiRoot` value, which is the object to which you should attach API endpoints, a `config` value, which is the server's configuration dictionary, and a `serverRoot` object, which can be used to attach endpoints that do not belong to the web API.

Within your plugin, you may import packages using relative imports or via the `girder.plugins` package. This will work for your own plugin, but you can also import modules from any active plugin. You can also import core Girder modules using the `girder` package as usual. Example:

```
from girder.plugins.cats import some_module
from girder import events
```

Adding a new route to the web API

If you want to add a new route to an existing core resource type, just call the `route()` function on the existing resource type. For example, to add a route for `GET /item/:id/cat` to the system,

```
from girder.api import access
from girder.api.rest import boundHandler

@access.public
@boundHandler()
def myHandler(self, id, params):
    self.requireParams('cat', params)

    return {
        'itemId': id,
        'cat': params['cat']
    }

def load(info):
    info['apiRoot'].item.route('GET', (':id', 'cat'), myHandler)
```

You should always add an access decorator to your handler function or method to indicate who can call the new route. The decorator is one of `@access.admin` (only administrators can call this endpoint), `@access.user` (any user who is logged in can call the endpoint), or `@access.public` (any client can call the endpoint).

In the above example, the *girder.api.rest.boundHandler* decorator is used to make the unbound method `myHandler` behave as though it is a member method of a *girder.api.rest.Resource* instance, which enables convenient access to methods like `self.requireParams`.

If you do not add an access decorator, a warning message appears: `WARNING: No access level specified for route GET item/:id/cat`. The access will default to being restricted to administrators.

When you start the server, you may notice a warning message appears: `WARNING: No description docs present for route GET item/:id/cat`. You can add self-describing API documentation to your route using the `describeRoute` decorator and `Description` class as in the following example:

```
from girder.api.describe import Description, describeRoute
from girder.api import access

@access.public
@describeRoute(
    Description('Retrieve the cat for a given item.')
    .param('id', 'The item ID', paramType='path')
    .param('cat', 'The cat value.', required=False)
    .errorResponse())
def myHandler(id, params):
    return {
        'itemId': id,
        'cat': params.get('cat', 'No cat param passed')
    }
```

That will make your route automatically appear in the Swagger documentation and will allow users to interact with it via that UI. See the *RESTful API docs* for more information about the Swagger page.

If you are creating routes that you explicitly do not wish to be exposed in the Swagger documentation for whatever reason, you can pass `None` to the `describeRoute` decorator, and no warning will appear.

```
@describeRoute(None)
```

Adding a new resource type to the web API

Perhaps for our use case we determine that `cat` should be its own resource type rather than being referenced via the `item` resource. If we wish to add a new resource type entirely, it will look much like one of the core resource classes,

and we can add it to the API in the `load()` method.

```
from girder.api.rest import Resource

class Cat(Resource):
    def __init__(self):
        super(Cat, self).__init__()
        self.resourceName = 'cat'

        self.route('GET', (), self.findCat)
        self.route('GET', (':id',), self.getCat)
        self.route('POST', (), self.createCat)
        self.route('PUT', (':id',), self.updateCat)
        self.route('DELETE', (':id',), self.deleteCat)

    def getCat(self, id, params):
        ...

def load(info):
    info['apiRoot'].cat = Cat()
```

Adding a new model type in your plugin

Most of the time, if you add a new resource type in your plugin, you'll have a `Model` class backing it. These model classes work just like the core model classes as described in the *Models* section. They must live under the `server/models` directory of your plugin, so that they can use the `ModelImporter` behavior. If you make a `Cat` model in your plugin, you could access it using

```
self.model('cat', 'cats')
```

Where the second argument to `model` is the name of your plugin.

The events system

In addition to being able to augment the core API as described above, the core system fires a known set of events that plugins can bind to and handle as they wish.

In the most general sense, the events framework is simply a way of binding arbitrary events with handlers. The events are identified by a unique string that can be used to bind handlers to them. For example, if the following logic is executed by your plugin at startup time,

```
from girder import events

def handler(event):
    print event.info

events.bind('some_event', 'my_handler', handler)
```

And then during runtime the following code executes:

```
events.trigger('some_event', info='hello')
```

Then `hello` would be printed to the console at that time. More information can be found in the API documentation for *Events*.

There are a specific set of known events that are fired from the core system. Plugins should bind to these events at `load` time. The semantics of these events are enumerated below.

- **Before REST call**

Whenever a REST API route is called, just before executing its default handler, plugins will have an opportunity to execute code or conditionally override the default behavior using `preventDefault` and `addResponse`. The identifiers for these events are of the form `rest.get.item/:id.before`. They receive the same kwargs as the default route handler in the event's info.

Since handlers of this event run prior to the normal access level check of the underlying route handler, they are bound by the same access level rules as route handlers; they must be decorated by one of the functions in *girder.api.access*. If you do not decorate them with one, they will default to requiring administrator access. This is to prevent accidental reduction of security by plugin developers. You may change the access level of the route in your handler, but you will need to do so explicitly by declaring a different decorator than the underlying route handler.

- **After REST call**

Just like the before REST call event, but this is fired after the default handler has already executed and returned its value. That return value is also passed in the event.info for possible alteration by the receiving handler. The identifier for this event is, e.g., `rest.get.item/:id.after`.

You may alter the existing return value, for example adding an additional property

```
event.info['returnVal']['myProperty'] = 'myPropertyValue'
```

or override it completely using `preventDefault` and `addResponse` on the event

```
event.addResponse(myReplacementResponse)
event.preventDefault()
```

- **Before model save**

You can receive an event each time a document of a specific resource type is saved. For example, you can bind to `model.folder.save` if you wish to perform logic each time a folder is saved to the database. You can use `preventDefault` on the passed event if you wish for the normal saving logic not to be performed.

- **After model creation**

You can receive an event *after* a resource of a specific type is created and saved to the database. This is sent immediately before the after-save event, but only occurs upon creation of a new document. You cannot prevent any default actions with this hook. The format of the event name is, e.g. `model.folder.save.created`.

- **After model save**

You can also receive an event *after* a resource of a specific type is saved to the database. This is useful if your handler needs to know the `_id` field of the document. You cannot prevent any default actions with this hook. The format of the event name is, e.g. `model.folder.save.after`.

- **Before model deletion**

Triggered each time a model is about to be deleted. You can bind to this via e.g., `model.folder.remove` and optionally `preventDefault` on the event.

- **During model copy**

Some models have a custom copy method (folder uses copyFolder, item uses copyItem). When a model is copied, after the initial record is created, but before associated models are copied, a copy.prepare event is sent, e.g. `model.folder.copy.prepare`. The event handler is passed a tuple of `((original model document), (copied model document))`. If the copied model is altered, the handler should save it without triggering events.

When the copy is fully complete, and copy.after event is sent, e.g. `model.folder.copy.after`.

- **Override model validation**

You can also override or augment the default `validate` methods for a core model type. Like the normal validation, you should raise a `ValidationException` for failure cases, and you can also `preventDefault` if you wish for the normal validation procedure not to be executed. The identifier for these events is, e.g., `model.user.validate`.

- **Override user authentication**

If you want to override or augment the normal user authentication process in your plugin, bind to the `auth.user.get` event. If your plugin can successfully authenticate the user, it should perform the logic it needs and then `preventDefault` on the event and `addResponse` containing the authenticated user document.

- **Before file upload**

This event is triggered as an upload is being initialized. The event `model.upload.assetstore` is sent before the `model.upload.save` event. The event information is a dictionary containing `model` and `resource` with the resource model type and resource document of the upload parent. For new uploads, the model type will be either `item` or `folder`. When the contents of a file are being replaced, this will be a `file`. To change from the current assetstore, add an `assetstore` key to the event information dictionary that contains an assetstore model document.

- **Just before a file upload completes**

The event `model.upload.finalize` after the upload is completed but before the new file is saved. This can be used if the file needs to be altered or the upload should be cancelled at the last moment.

- **On file upload**

This event is always triggered asynchronously and is fired after a file has been uploaded. The file document that was created is passed in the event info. You can bind to this event using the identifier `data.process`.

- **Before file move**

The event `model.upload.movefile` is triggered when a file is about to be moved from one assetstore to another. The event information is a dictionary containing `file` and `assetstore` with the current file document and the target assetstore document. If `preventDefault` is called, the move will be cancelled.

Note: If you anticipate your plugin being used as a dependency by other plugins, and want to potentially alert them of your own events, it can be worthwhile to trigger your own events from within the plugin. If you do that, the identifiers for those events should begin with the name of your plugin, e.g., `events.trigger('cats.something_happened', info='foo')`

Automated testing for plugins

Girder makes it easy to add automated testing to your plugin that integrates with the main Girder testing framework. In general, any CMake code that you want to be executed for your plugin can be performed by adding a **plugin.cmake** file in your plugin.

```
cd plugins/cats ; touch plugin.cmake
```

That file will be automatically included when Girder is configured by CMake. To add tests for your plugin, you can make use of some handy CMake functions provided by the core system. For example:

```
add_python_test(cat PLUGIN cats)
add_python_style_test(python_static_analysis_cats "${PROJECT_SOURCE_DIR}/plugins/cats/server")
```

Then you should create a `plugin_tests` package in your plugin:

```
mkdir plugin_tests ; cd plugin-tests ; touch __init__.py cat_test.py
```

The **cat_test.py** file should look like:

```
from tests import base


def setUpModule():
    base.enabledPlugins.append('cats')
    base.startServer()


def tearDownModule():
    base.stopServer()


class CatsCatTestCase(base.TestCase):

    def testCatsWork(self):
        ...
```

You can use all of the testing utilities provided by the `base.TestCase` class from core. You will also get coverage results for your plugin aggregated with the main Girder coverage results if coverage is enabled.

Plugins can also use the external data interface provided by Girder as described in *Downloading external data files for test cases*. For plugins, the data key files should be placed inside a directory called `plugin_tests/data/`. When referencing the files, they must be prefixed by your plugin name as follows

```
add_python_test(my_test EXTERNAL_DATA plugins/cats/test_file.txt)
```

Then inside your unittest, the file will be available under the main data path as `os.environ['GIRDER_TEST_DATA_PREFIX'] + '/plugins/cats/test_file.txt'`.

Extending the Client-Side Application

The web client may be extended independently of the server side. Plugins may import Pug templates, Stylus files, and JavaScript files into the application. The plugin loading system ensures that only content from enabled plugins gets loaded into the application at runtime.

All of your plugin's extensions to the web client must live in a directory in the top level of your plugin called **web_client**.

```
cd plugins/cats ; mkdir web_client
```

Under the **web_client** directory, you must have a webpack entry point file called **main.js**. In this file, you can import code from your plugin using relative paths, or relative to the special alias **girder_plugins/<your_plugin_key>**. For example, `import template from 'girder_plugins/cats/templates/myTemplate.pug` would import the template file located at `plugins/cats/web_client/templates/myTemplate.pug`. Core Girder code can be imported relative to the path **girder**, for example `import View from 'girder/views/View';`. The entry point defined in your **main.js** file will be automatically built once the plugin has been enabled, and your built code will be served with the application once the server has been restarted.

Linting and Style Checking Client-Side Code

Girder uses ESLint to perform static analysis of its own JavaScript files. Developers can easily add the same static analysis tests to their own plugins using a CMake function call defined by Girder.

```
add_eslint_test(
    js_static_analysis_cats "${PROJECT_SOURCE_DIR}/plugins/cats/web_client"
)
```

This will check all files with the extension **.js** inside of the `cats` plugin's `web_client` directory using the same style rules enforced within Girder itself. Plugin developers can also choose to extend or even override entirely the core style rules. To do this, you only need to provide a path to a custom ESLint configuration file as follows.

```
add_eslint_test(
    js_static_analysis_cats "${PROJECT_SOURCE_DIR}/plugins/cats/web_client"
    ESLINT_CONFIG_FILE "${PROJECT_SOURCE_DIR}/plugins/cats/.eslintrc"
)
```

You can configure ESLint inside this file however you choose. For example, to extend Girder's own configuration by adding a new global variable `cats` and you really hate using semicolons, you can put the following in your **.eslintrc**

```
{
    "extends": "../../.eslintrc",
    "globals": {
        "cats": true
    },
    "rules": {
        "semi": 0
    }
}
```

You can also lint your pug templates using the `pug-lint` tool.

```
add_puglint_test(cats "${PROJECT_SOURCE_DIR}/plugins/cats/web_client/templates")
```

Installing custom dependencies from npm

There are two types of node dependencies you may need to install for your plugin. Each type needs to be installed differently due to how node manages external packages.

- Run time dependencies that your application relies on may be handled in one of two ways. If you are writing a simple plugin that does not contain its own Gruntfile, these dependencies should be installed into Girder's own **node_modules** directory by specifying them in the `npm.dependencies` section of your `plugin.json` file.

```
    {
        "name": "MY_PLUGIN",
        "npm": {
            "dependencies": {
                "vega": "^2.6.0"
            }
        }
    }
```

  If instead you are using a custom Grunt build with a Gruntfile, the dependencies should be installed into your plugin's **node_modules** directory by providing a package.json file just as they are used for standalone node applications. When such a file exists in your plugin directory, `npm install` will be executed in a new process from within your package's directory.

- Build time dependencies that your Grunt tasks rely on to assemble the sources for deployment need to be installed into Girder's own **node_modules** directory. These dependencies will typically be Grunt extensions defining extra tasks used by your build. Such dependencies should be listed under `grunt.dependencies` as an object (much like dependencies in **package.json**) inside your **plugin.json** or **plugin.yml** file.

```
    {
        "name": "MY_PLUGIN",
        "grunt": {
```

```
            "dependencies": {
                "grunt-shell": ">=0.2.1"
            }
        }
    }
```

In addition to installing these dependencies, Girder will also load grunt extensions contained in them before executing any tasks.

Note: Packages installed into Girder's scope can possibly overwrite an alternate version of the same package. Care should be taken to only list packages here that are not already provided by Girder's own build time dependencies.

Executing custom Grunt build steps for your plugin

For more complex plugins which require custom Grunt tasks to build, the user can specify custom targets within their own Grunt file that will be executed when the main Girder Grunt step is executed. To use this functionality, add a **grunt** key to your **plugin.json** file.

```
{
"name": "MY_PLUGIN",
"grunt":
    {
    "file" : "Gruntfile.js",
    "defaultTargets": [ "MY_PLUGIN_TASK" ],
    "autobuild": true
    }
}
```

This will allow to register a Gruntfile relative to the plugin root directory and add any target to the default one using the "defaultTargets" array.

Note: The **file** key within the **grunt** object must be a path that is relative to the root directory of your plugin. It does not have to be called `Gruntfile.js`, it can be called anything you want.

Note: Girder creates a number of Grunt build tasks that expect plugins to be organized according to a certain convention. To opt out of these tasks, add an **autobuild** key (default: **true**) within the **grunt** object and set it to **false**.

All paths within your custom Grunt tasks must be relative to the root directory of the Girder source repository, rather than relative to the plugin directory.

```
module.exports = function (grunt) {
    grunt.registerTask('MY_PLUGIN_TASK', 'Custom plugin build task', function () {
        /* ... Execute custom behavior ... */
    });
};
```

JavaScript extension capabilities

Plugins may bind to any of the normal events triggered by core via a global events object that can be imported like so:

```
import events from 'girder/events';

...

this.listenTo(events, 'g:event_name', () => { do.something(); });
```

This will accommodate certain events, such as before and after the application is initially loaded, and when a user logs in or out, but most of the time plugins will augment the core system using the power of JavaScript rather than the explicit events framework. One of the most common use cases for plugins is to execute some code either before or after one of the core model or view functions is executed. In an object-oriented language, this would be a simple matter of extending the core class and making a call to the parent method. The prototypal nature of JavaScript makes that pattern impossible; instead, we'll use a slightly less straightforward but equally powerful mechanism. This is best demonstrated by example. Let's say we want to execute some code any time the core `HierarchyWidget` is rendered, for instance to inject some additional elements into the view. We use Girder's `wrap` utility function to *wrap* the method of the core prototype with our own function.

```
import HierarchyWidget from 'girder/views/widgets/HierarchyWidget';
import { wrap } from 'girder/utilities/PluginUtils';

// Import our template file from our plugin using a relative path
import myTemplate from './templates/hierachyWidgetExtension.pug';

// CSS files pertaining to this view should be imported as a side-effect
import './stylesheets/hierarchyWidgetExtension.styl';

wrap(HierarchyWidget, 'render', function (render) {
    // Call the underlying render function that we are wrapping
    render.call(this);

    // Add a link just below the widget using our custom template
    this.$('.g-hierarchy-widget').after(myTemplate());
});
```

Notice that instead of simply calling `render()`, we call `render.call(this)`. That is important, as otherwise the value of `this` will not be set properly in the wrapped function.

Now that we have added the link to the core view, we can bind an event handler to it to make it functional:

```
HierarchyWidget.prototype.events['click a.cat-link'] = () => {
    alert('meow!');
};
```

This demonstrates one simple use case for client plugins, but using these same techniques, you should be able to do almost anything to change the core application as you need.

Setting an empty layout for a route

If you have a route in your plugin that you would like to have an empty layout, meaning that the Girder header, nav bar, and footer are hidden and the Girder body is evenly padded and displayed, you can specify an empty layout in the `navigateTo` event trigger.

As an example, say your plugin wanted a `frontPage` route for a Collection which would display the Collection with only the Girder body shown, you could add the following route to your plugin.

```
import events from 'girder/events';
import router from 'girder/router';
import { Layout } from 'girder/constants';
```

```
import CollectionModel from 'girder/models/CollectionModel';
import CollectionView from 'girder/views/body/CollectionView';

router.route('collection/:id/frontPage', 'collectionFrontPage', function (collectionId, params) {
    var collection = new CollectionModel();
    collection.set({
        _id: collectionId
    }).on('g:fetched', function () {
        events.trigger('g:navigateTo', CollectionView, _.extend({
            collection: collection
        }, params || {}), {layout: Layout.EMPTY});
    }, this).on('g:error', function () {
        router.navigate('/collections', {trigger: true});
    }, this).fetch();
});
```

3.3.4 Developer Cookbook

This cookbook consists of a set of examples of common tasks that developers may encounter when developing Girder applications.

Client cookbook

The following examples are for common tasks that would be performed by a Girder client application.

Authenticating to the web API

Clients can make authenticated web API calls by passing a secure temporary token with their requests. Tokens are obtained via the login process; the standard login process requires the client to make an HTTP `GET` request to the `api/v1/user/authentication` route, using HTTP Basic Auth to pass the user credentials. For example, for a user with login "john" and password "hello", first base-64 encode the string `"john:hello"` which yields `"am9objpoZWxsbw=="`. Then take the base-64 encoded value and pass it via the `Girder-Authorization` header (The `Authorization` header will also work):

```
Girder-Authorization: Basic am9objpoZWxsbw==
```

If the username and password are correct, you will receive a 200 status code and a JSON document from which you can extract the authentication token, e.g.:

```
{
  "authToken": {
    "token": "urXQSHO8aF6cLB5si0Ch0WCiblvW1m8YSFylMH9eqN1Mt9KvWUnghVDKQy545ZeA",
    "expires": "2015-04-11 00:06:14.598570"
  },
  "message": "Login succeeded.",
  "user": {
    ...
  }
}
```

The `authToken.token` string is the token value you should pass in subsequent API calls, which should either be passed as the `token` parameter in the query or form parameters, or as the value of a custom HTTP header with the key `Girder-Token`, e.g.

```
Girder-Token: urXQSHO8aF6cLB5si0Ch0WCiblvW1m8YSFylMH9eqN1Mt9KvWUnghVDKQy545ZeA
```

Note: When logging in, the token is also sent to the client in a Cookie header so that web-based clients can persist its value conveniently for its duration. However, for security reasons, merely passing the cookie value back is not sufficient for authentication.

Note: If you are using Girder's JavaScript web client library in a CORS environment, be sure to set `girder.corsAuth = true;` in your application prior to calling `girder.login`. This will allow users' login sessions to be saved on the origin site's cookie.

Upload a file

If you are using the Girder javascript client library, you can simply call the `upload` method of the `girder/models/FileModel`. The first argument is the parent model object (an `ItemModel` or `FolderModel` instance) to upload into, and the second is a browser `File` object that was selected via a file input element. You can bind to several events of that model, as in the example below.

```
import FileModel from 'girder/models/FileModel';

var fileModel = new FileModel();
fileModel.on('g:upload.complete', function () {
    // Called when the upload finishes
}).on('g:upload.chunkSent', function (info) {
    // Called on each chunk being sent
}).on('g:upload.progress', function (info) {
    // Called regularly with progress updates
}).on('g:upload.error', function (info) {
    // Called if an upload fails partway through sending the data
}).on('g:upload.errorStarting', function (info) {
    // Called if an upload fails to start
});
fileModel.upload(parentFolder, fileObject);
```

If you don't feel like making your own upload interface, you can simply use the `girder/views/widgets/UploadWidget` to provide a nice GUI interface for uploading. It will prompt the user to drag and drop or browse for files, and then shows a current and overall progress bar and also provides controls for resuming a failed upload.

Using the Girder upload widget in a custom app

Your custom javascript application can easily reuse the existing upload widget provided in the Girder javascript library if you don't want to write your own upload view. This can save time spent duplicating functionality, since the upload widget provides current and overall progress bars, file displays, a drag-and-droppable file selection button, resume behavior in failure conditions, and customizable hooks for various stages of the upload process.

The default behavior of the upload widget is to display as a modal dialog, but many users will want to simply embed it underneath a normal DOM element flow. The look and behavior of the widget can be customized when the widget is instantiated by passing in options like so:

```
import UploadWidget from 'girder/views/widgets/UploadWidget';
```

```
new UploadWidget({
    option: value,
    ...
});
```

The following options are not required, but may be used to modify the behavior of the widget:

- `[parent]` - If the parent object is known when instantiating this upload widget, pass the object here.
- `[parentType=folder]` - If the parent type is known when instantiating this upload widget, pass the object here. Otherwise set `noParent: true` and set it later, prior to starting the upload.
- `[noParent=false]` - If the parent object being uploaded into is not known at the time of widget instantiation, pass `noParent: true`. Callers must ensure that the parent is set by the time `uploadNextFile()` actually gets called.
- `[title="Upload files"]` - Title for the widget. This is highly recommended when rendering as a modal dialog. To disable rendering of the title, simply pass a falsy object.
- `[modal=true]` - This widget normally renders as a modal dialog. Pass `modal: false` to disable the modal behavior and simply render underneath a parent element.
- `[overrideStart=false]` - Some callers will want to hook into the pressing of the start upload button and add their own logic prior to actually sending the files. To do so, set `overrideStart: true` and bind to the `g:uploadStarted` event of this widget. The caller is then responsible for calling `uploadNextFile()` on the widget when they have completed their actions and are ready to actually send the files.

For general documentation on embedding Girder widgets in a custom application, see the section on *client development*.

Server cookbook

The following examples refer to tasks that are executed by the Girder application server.

Creating a REST route

The process of creating new REST resources and routes is documented *here*.

The API docs of the `route` method can be found here.

Loading a resource by its ID

This is a fundamental element of many REST operations; they receive a parameter representing a resource's unique ID, and want to load the corresponding resource from that ID. This behavior is known as model loading. As a brief example, if we had the ID of a folder within our REST route handler, and wanted to load its corresponding document from the database, it would look like:

```
self.model('folder').load(theFolderId, user=self.getCurrentUser(), level=AccessType.READ)
```

The load method of each model class takes the resource's unique ID as its first argument (this is the `_id` field in the documents). For access controlled models like the above example, it also requires the developer to specify which user is requesting the loading of the resource, and what access level is required on the resource. If the ID passed in does not correspond to a record in the database, `None` is returned.

Sometimes models need to be loaded outside the context of being requested by a specific user, and in those cases the `force` flag should be used:

```
self.model('folder').load(theFolderId, force=True)
```

If you need to load a model that is in a plugin rather than a core model, pass the plugin name as the second argument to the `model` method:

```
self.model('cat', 'cats').load(...)
```

The ModelImporter class conveniently exposes a method for retrieving instances of models that are statically cached for efficient reuse. You can mix this class into any of your classes to enable `self.model` semantics. The `ModelImporter.model` method is static, so you can also just do the following anywhere:

```
ModelImporter.model('folder')...
```

Send a raw or streaming HTTP response body

For consistency, the default behavior of a REST endpoint in Girder is to take the return value of the route handler and encode it in the format specified by the client in the `Accepts` header, usually `application/json`. However, in some cases you may want to force your endpoint to send a raw response body back to the client.

If you want to send a raw response, you can simply decorate your route handler with the `girder.api.rest.rawResponse` decorator, or call `girder.api.rest.setRawResponse()` within the body of the route handler. For example:

```
from girder.api import access, rest

@access.public
@rest.rawResponse
def rawExample(self, params):
    return 'raw string'
```

That will make the response body precisely the string `raw string`. If the data size being sent to the client is large or unbounded, you should use a streaming response.

If you want to send a streaming response, simply make your route handler return a generator function. A streaming response is automatically sent as a raw response. Developers have full control of the buffer size of the streamed response body; each time you `yield` data in your generator function, the buffer will be flushed to the client. As a minimal example, the following route handler would flush 10 chunks to the client, and the full response body would be `0123456789`.

```
from girder.api import access

@access.public
def streamingExample(self, params):
    def gen():
        for i in range(10):
            yield str(i)
    return gen
```

Serving a static file

If you are building a plugin that needs to serve up a static file from a path on disk, you can make use of the `staticFile` utility, as in the following example:

```
import os
from girder.utility.server import staticFile
```

```
def load(info):
    path = os.path.join(PLUGIN_ROOT_DIR, 'static', 'index.html')
    info['serverRoot'].static_route = staticFile(path)
```

The `staticFile` utility should be assigned to the route corresponding to where the static file should be served from.

Note: If a relative path is passed to `staticFile`, it will be interpreted relative to the current working directory, which may vary. If your static file resides within your plugin, it is recommended to use the special `PLUGIN_ROOT_DIR` property of your server module, or the equivalent `info['pluginRootDir']` value passed to the `load` method.

Sending Emails

Girder has a utility module that make it easy to send emails from the server. For the sake of maintainability and reusability of the email content itself, emails are stored as Mako templates in the **girder/mail_templates** directory. By convention, email templates should include `_header.mako` above and `_footer.mako` below the content. If you wish to send an email from some point within the application, you can use the utility functions within `girder.utility.mail_utils`, as in the example below:

```
from girder.utility import mail_utils

...

def my_email_sending_code():
    html = mail_utils.renderTemplate('myContentTemplate.mako', {
        'param1': 'foo',
        'param2': 'bar'
    })
    mail_utils.sendEmail(to=email, subject='My mail from Girder', text=html)
```

If you wish to send email from within a plugin, simply create a **server/mail_templates** directory within your plugin, and it will be automatically added to the mail template search path when your plugin is loaded. To avoid name collisions, convention dictates that mail templates within your plugin should be prefixed by your plugin name, e.g., `my_plugin.my_template.mako`.

If you want to send email to all of the site administrators, there is a convenience keyword argument for that. Rather than setting the `to` field, pass `toAdmins=True`.

```
mail_utils.sendEmail(toAdmins=True, subject='...', text='...')
```

Note: All emails are sent as rich text (`text/html` MIME type).

Logging a Message

Girder application servers maintain an error log and an information log and expose a utility module for sending events to them. Any 500 error that occurs during execution of a request will automatically be logged in the error log with a full stack trace. Also, any 403 error (meaning a user who is logged in but requests access to a resource that they don't have permission to access) will also be logged automatically. All log messages automatically include a timestamp, so there is no need to add your own.

If you want to log your own custom error or info messages outside of those default behaviors, use the following examples:

```
from girder import logger

try:
    ...
except Exception:
    # Will log the most recent exception, including a traceback, request URL,
    # and remote IP address. Should only be called from within an exception handler.
    logger.exception('A descriptive message')

# Will log a message to the info log.
logger.info('Test')
```

Adding Automated Tests

The server side Python tests are run using unittest. All of the actual test cases are stored under *tests/cases*.

Adding to an Existing Test Case

If you want to add tests to an existing test case, just create a new function in the relevant TestCase class. The function name must start with **test**. If the existing test case has **setUp** or **tearDown** methods, be advised that those methods will be run before and after *each* of the test methods in the class.

Creating a New Test Case

To create an entirely new test case, create a new file in **cases** that ends with **_test.py**. To start off, put the following code in the module (with appropriate class name of course):

```
from .. import base

def setUpModule():
    base.startServer()

def tearDownModule():
    base.stopServer()

class MyTestCase(base.TestCase):
```

Note: If your test case does not need to communicate with the server, you do not need to call **base.startServer()** and **base.stopServer()** in the **setUpModule()** and **tearDownModule()** functions. Those functions are called once per module rather than once per test method.

Then, in the **MyTestCase** class, just add functions that start with **test**, and they will automatically be run by unittest.

Finally, you'll need to register your test in the *CMakeLists.txt* file in the *tests* directory. Just add a line like the ones already there at the bottom. For example, if the test file you created was called *thing_test.py*, you would add:

```
add_python_test(thing)
```

Re-run CMake in the build directory, and then run CTest, and your test will be run.

Note: By default, **add_python_test** allows the test to be run in parallel with other tests, which is normally fine since each python test has its own assetstore space and its own mongo database, and the server is typically mocked rather than actually binding to its port. However, some tests (such as those that actually start the cherrypy server) should not be run concurrently with other tests that use the same resource. If you have such a test, use the `RESOURCE_LOCKS` argument to **add_python_test**. If your test requires the cherrypy server to bind to its port, declare that it locks the `cherrypy` resource. If it also makes use of the database, declare that it locks the `mongo` resource. For example:

```
add_python_test(my_test RESOURCE_LOCKS cherrypy mongo)
```

Downloading external data files for test cases

In some cases, it is necessary to perform a test on a file that is too big store inside a repository. For tests such as these, Girder provides a way to link to test files served at https://midas3.kitware.com and have them automatically downloaded and cached during the build stage. To add a new external file, first make an account at https://midas3.kitware.com and upload a publicly accessible file. When viewing the items containing those files on Midas, there will be a link to "Download key file" appearing as a key icon. This file contains the MD5 hash of the file contents and can be committed inside the `tests/data/` directory of Girder's repository. This file can then be listed as an optional `EXTERNAL_DATA` argument to the `add_python_test` function to have the file downloaded as an extra build step. As an example, consider the file currently used for testing called `tests/data/test_file.txt.md5`. To use this file in you test, you would add the test as follows

```
add_python_test(my_test EXTERNAL_DATA test_file.txt)
```

The `EXTERNAL_DATA` keyword argument can take a list of files or even directories. When a directory is provided, it will download all files that exist in the given path. Inside your unit test, you can access these files under the path given by the environment variable `GIRDER_TEST_DATA_PREFIX` as follows

```
import os
test_file = os.path.join(
    os.environ['GIRDER_TEST_DATA_PREFIX'],
    'test_file.txt'
)
with open(test_file, 'r') as f:
    content = f.read() # The content of the downloaded test file
```

Mounting a custom application

Normally, the root node (`/`) of the server will serve up the Girder web client. A plugin may contain an entire application separate from the default Girder web client. This plugin may be written in a way which enables administrators to mount the application at a configured endpoint, including the option of replacing the root node with the plugin application.

To achieve this, you simply have to register your own root and configure your routes as you wish. In your plugin's `load` method, you would follow this convention:

```
from girder.utility.plugin_utilities import registerPluginWebroot
registerPluginWebroot(CustomAppRoot(), info['name'])
```

This will register your `CustomAppRoot` with Girder so that it can then be mounted wherever an Administrator specifies using the Server Configuration Panel. See *Managing Routes*.

Supporting web browser operations where custom headers cannot be set

Some aspects of the web browser make it infeasible to pass the usual `Girder-Token` authentication header when making a request. For example, if using an `EventSource` object for SSE, or when you must redirect the user's browser to a download endpoint that serves its content as an attachment.

In such cases, you may allow specific REST API routes to authenticate using the Cookie. To avoid vulnerabilities to Cross-Site Request Forgery attacks, you should only do this if the endpoint is "read-only" (that is, the endpoint does not make modifications to data on the server). Accordingly, only routes for `HEAD` and `GET` requests allow cookie authentication to be enabled (without an additional override).

In order to allow cookie authentication for your route, simply add the `cookie` decorator to your route handler function. Example:

```
from girder.api import access

@access.cookie
@access.public
def download(self, params):
    ...
```

As a last resort, if your endpoint is not read-only and you are unable to pass the `Girder-Token` header to it, you can pass a `token` query parameter containing the token , but in practice this will probably never be the case.

3.3.5 External Web Clients

You may want to build your own custom web applications using Girder. Since Girder cleanly separates API from UI, it is straightforward to use a mounted Girder API for app authentication and data storage. You may additionally use Girder's JavaScript libraries and UI templates to assist in building applications.

Including the Girder REST API

Apache

See the *Deploy* section for instructions on deployment of Girder under Apache. You may host your web application alongside Girder and use its REST interface.

Tangelo

Tangelo is a CherryPy based web server framework for rapid data analytics and visualization application development. Tangelo has options for directly mounting the Girder API and static application files inside a Tangelo instance. See details in Tangelo's setup documentation.

Using Girder JavaScript Utilities and Views

Including the JavaScript

Use the following to include the Girder libraries in your web application, assuming Girder is hosted at `/girder`:

```
<script src="/girder/static/built/girder.ext.min.js"></script>
<script src="/girder/static/built/girder.app.min.js"></script>
```

Note: `girder.ext.min.js` includes requirements for Girder, including jQuery, Bootstrap, Underscore, and Backbone. You may wish to use your own versions of these separately and not include `girder.ext.min.js`.

Extending Girder's Backbone application

Girder defines a top-level class at `girder.App`. This object is responsible for bootstraping the application, setting up the overall layout, and responding to global events like `g:login` and `g:navigateTo`. Developers can choose

to derive their own application from this class to use the functionality that it provides. For example, the following derivation would modify the normal application bootstrapping

```
// set the path where girder's API is mounted
girder.apiRoot = '/girder/api/v1';

var App = girder.App.extend({
   start: function () {

      // disable girder's router
      girder.router.enabled(false);

      // call the super method
      return girder.App.prototype.start.call(this, {
          fetch: false,  // disable automatic fetching of the user model
          history: false,// disable initialization of Backbone's router
          render: false  // disable automatic rendering on start
      }).then(_.bind(function () {

         // set the current user somehow
         girder.currentUser = new girder.models.UserModel({...});
         girder.eventStream.open();

         // replace the header with a customized class
         this.headerView = new MyHeaderView({parentView: this});

         // render the main page
         this.render();

         // start up the router with the `pushState` option enabled
         Backbone.history.start({pushState: true});
      }, this));
   }
});

// initialize the application without starting it
var app = new App({start: false});

// start your application after the page loads
$(document).ready(function () {
   app.start();
});
```

Other methods that one may need to override include the following:

`bindGirderEvents` Bind handlers to the global `girder.events` object.

`render` Render (or re-render) the entire page.

Note: `girder.router.enabled(false)` must be set to false to disable URL routing behavior specific to the full Girder web application.

Using Girder Register and Login UI

To use Girder UI components, you will need the following CSS files in your HTML:

```
<link rel="stylesheet" href="/girder/static/built/girder.ext.min.css">
<link rel="stylesheet" href="/girder/static/built/girder.app.min.css">
```

Note: `girder.ext.min.css` includes requirements for Girder, including Bootstrap and some additional Bootstrap extensions. You may wish to use your own versions of these separately and not include `girder.ext.min.css`.

To make login and logout controls, provide a dialog container and login/logout/register links, and a container where the dialogs will be rendered:

```
<button class="btn btn-link" id="login" href="#">Login</button>
<button class="btn btn-link" id="register" href="#">Register</button>
<label class="hidden" id="name" href="#"></label>
<button class="btn btn-link hidden" id="logout" href="#">Logout</button>
<div class="modal fade" id="dialog-container"></div>
```

In your JavaScript, perform callbacks such as the following:

```
$('#login').click(function () {
    var loginView = new girder.views.LoginView({
        el: $('#dialog-container')
    });
    loginView.render();
});

$('#register').click(function () {
    var registerView = new girder.views.RegisterView({
        el: $('#dialog-container')
    });
    registerView.render();
});

$('#logout').click(function () {
    girder.restRequest({
        path: 'user/authentication',
        type: 'DELETE'
    }).done(function () {
        girder.currentUser = null;
        girder.events.trigger('g:login');
    });
});

girder.events.on('g:login', function () {
    console.log("g:login");
    if (girder.currentUser) {
        $("#login").addClass("hidden");
        $("#register").addClass("hidden");
        $("#name").removeClass("hidden");
        $("#logout").removeClass("hidden");
        $("#name").text(girder.currentUser.get('firstName') + " " + girder.currentUser.get('lastName

        // Do anything else you'd like to do on login.
    } else {
        $("#login").removeClass("hidden");
        $("#register").removeClass("hidden");
        $("#name").addClass("hidden");
        $("#logout").addClass("hidden");
```

```
        // Do anything else you'd like to do on logout.
    }
});

// Check for who is logged in initially
girder.restRequest({
    path: 'user/authentication',
    error: null
}).done(function (resp) {
    girder.currentUser = new girder.models.UserModel(resp.user);
    girder.events.trigger('g:login');
});
```

You can find an example minimal application using Girder's login and register dialogs in the source tree at **/clients/web-external**.

3.3.6 Python Client and Girder CLI

In addition to the web clients, Girder comes with a python client library and a CLI to allow for programmatic interaction with a Girder server, and also to workaround limitations of the web client. For example, the python CLI makes it much easier to upload a large, nested hierarchy of data from a local directory to Girder, and also makes it much easier to download a large, nested hierarchy of data from Girder to a local directory.

Installation

If you have the source directory of Girder, you can find the `girder_client` package within the `clients/python` directory. If you do not have the source directory of Girder, you can install the client via pip:

```
pip install girder-client
```

The Command Line Interface

The girder_client package ships with a command-line utility that wraps some of its common functionality to make it easy to invoke operations without having to write any custom python scripts. If you have installed girder_client via pip, you can use the special `girder-cli` executable:

```
girder-cli <arguments>
```

Otherwise you can equivalently just invoke the module directly:

```
python -m girder_client <subcommand> <arguments>
```

To see all available subcommands, run:

```
girder-cli --help
```

For help with a specific subcommand, run:

```
girder-cli <subcommand> --help
```

Specifying the Girder Instance

When constructing a Girder client, you must declare what instance of Girder you wish to connect to. The easiest way to do so is to pass the full URL to the REST API of the Girder instance you wish to connect to using the `api-url` argument to `girder-cli`. For example:

```
girder-cli --api-url http://localhost:8080/api/v1 <subcommand> ...
```

You may also specify the URL in parts, using the `host` argument, and optional `scheme`, `port`, and `api-root` args.

```
girder-cli --host girder.example.com ...
```

Or...

```
girder-cli --host girder.example.com --scheme https --port 443 --api-root /api/v1 ...
```

Upload a local file hierarchy

To upload a folder hierarchy rooted at *test_folder* to the Girder Folder with id *54b6d41a8926486c0cbca367*

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder
```

When using the upload command, the default `--parent-type`, meaning the type of resource the local folder will be created under in Girder, is Folder, so the following are equivalent

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder
girder-cli upload 54b6d41a8926486c0cbca367 test_folder --parent-type folder
```

To upload that same local folder to a Collection or User, specify the parent type as follows

```
girder-cli upload 54b6d41a8926486c0cbca459 test_folder --parent-type user
```

To see what local folders and files on disk would be uploaded without actually uploading anything, add the `--dry-run` flag

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder --dry-run
```

To have leaf folders (those folders with no subfolders, only containing files) be uploaded to Girder as single Items with multiple Files, i.e. those leaf folders will be created as Items and all files within the leaf folders will be Files within those Items, add the `--leaf-folders-as-items` flag

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder --leaf-folders-as-items
```

If you already have an existing Folder hierarchy in Girder which you have a superset of on your local disk (e.g. you previously uploaded a hierarchy to Girder and then added more folders and files to the hierarchy on disk), you can reuse the existing hierarchy in Girder, which will not create new Folders and Items for those that match folders and files on disk, by using the `--reuse` flag.

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder --reuse
```

To include a blacklist of file patterns that will not be uploaded, pass a comma separated list to the `--blacklist` arg

```
girder-cli upload 54b6d41a8926486c0cbca367 test_folder --blacklist .DS_Store
```

Download a Folder hierarchy into a local folder

To download a Girder Folder hierarchy rooted at Folder id *54b6d40b8926486c0cbca364* under the local folder *download_folder*

```
girder-cli download 54b6d40b8926486c0cbca364 download_folder
```

Downloading is only supported from a parent type of Folder.

Synchronize local folder with a Folder hierarchy

If the *download_folder* is a local copy of a Girder Folder hierarchy rooted at Folder id *54b6d40b8926486c0cbca364*, any change made to the Girder Folder remotely can be synchronized locally by

```
girder-cli localsync 54b6d40b8926486c0cbca364 download_folder
```

This will only download new Items or Items that have been modified since the last download/localsync. Local files that are no longer present in the remote Girder Folder will not be removed. This command relies on a presence of metadata file *.metadata-girder* within *download_folder*, which is created upon *girder-cli download*. If *.metadata-girder* is not present, *localsync* will fallback to *download*.

The Python Client Library

For those wishing to write their own python scripts that interact with Girder, we recommend using the Girder python client library, documented below.

Recursively inherit access control to a Folder's descendants

This will take the access control and public value in the Girder Folder with id *54b43e9b8926486c0c06cb4f* and copy those to all of the descendant Folders

```
import girder_client
gc = girder_client.GirderClient(apiUrl='https://data.kitware.com/api/v1')
gc.authenticate('username', 'password')
gc.inheritAccessControlRecursive('54b43e9b8926486c0c06cb4f')
```

Set callbacks for Folder and Item uploads

If you have a function you would like called upon the completion of an Item or Folder upload, you would do the following.

N.B. The Item callbacks are called after the Item is created and all Files are uploaded to the Item. The Folder callbacks are called after the Folder is created and all child Folders and Items are uploaded to the Folder.

```
import girder_client
gc = girder_client.GirderClient()

def folderCallback(folder, filepath):
    # assume we have a folderMetadata dict that has
    # filepath: metadata_dict_for_folder
    gc.addMetadataToFolder(folder['_id'], folderMetadata[filepath])

def itemCallback(item, filepath):
```

```
    # assume we have an itemMetadata dict that has
    # filepath: metadata_dict_for_item
    gc.addMetadataToItem(item['_id'], itemMetadata[filepath])

gc.authenticate('username', 'password')
gc.addFolderUploadCallback(folderCallback)
gc.addItemUploadCallback(itemCallback)
gc.upload(localFolder, parentId)
```

Further Examples and Function Level Documentation

3.3.7 Security

Girder maintains data security through a variety of mechanisms.

Default Authorization

Internally, endpoints default to requiring administrator permissions in order to use them. This means that, for example, when writing a plugin, a developer must consciously choose to allow non-administrator access. Basic administrator, user, or token access restrictions are applied before any other endpoint code is executed.

CORS (Cross-Origin Resource Sharing)

In an out-of-the-box Girder deployment, CORS is disabled for API calls. If you want your server to support API calls that are cross-origin requests from web browsers, you'll need to modify some configuration settings.

As an administrator, go to the **Admin console**, then to **Server configuration**. Open the **Advanced Settings** panel and you will see several settings that allow you to specify the CORS policies for the REST API. The most important setting is the **CORS Allowed Origins** field, which is used to specify what origins are allowed to make cross-origin requests to the instance's REST API. By default, this is blank, meaning no cross-origin requests are allowed. To allow requests from *any* origin, simply set this to `*`. Whatever you set here will be passed back to the browser in the `Access-Control-Allow-Origin` header, which the browser uses to allow or deny the cross-origin request.

If you want more fine-grained control over the CORS policies, you can also restrict the allowed methods and allowed request headers by providing them in comma-separated lists in the **CORS Allowed Methods** and **CORS Allowed Headers** fields, though this is usually not necessary–the default values for these two fields are quite permissive and should enable complete access to the web API so long as the origin is allowed.

These settings simply control the CORS headers that are sent to the browser; actual enforcement of the CORS policies takes place on the user's browser.

Database Injection Attacks

Girder defends against database injection attacks by using PyMongo as the only pathway between the application server and the database server. This protects against many injection vulnerabilities as described in the MongoDB Documentation. Girder also uses a model layer to mediate and validate all interaction with the database. This ensures that for all database operations, structural attributes (collection name, operation type, etc.) are hardcoded and not modifiable by the client, while data attributes (stored content) are validated for proper form before being accepted from a client.

Additionally, we strongly recommend configuring your MongoDB server with JavaScript disabled unless explicitly needed for your Girder-based application or plugin. Again, see the MongoDB Documentation for more information.

Session Management

Girder uses session management performed through the Girder-Token header or through a token passed through a GET parameter. This token is provided to the client through the cookie and expires after a configurable amount of time. In order to prevent session stealing, it is highly recommended to run Girder under HTTPS.

Cross-Site Scripting (XSS)

In order to protect against XSS attacks, all input from users is sanitized before presentation of the content on each page. This is handled by the template system Girder uses (Pug). This sanitizes user-provided content.

Cross-Site Request Forgery (CSRF)

To prevent CSRF attacks, Girder requires the Girder-Token parameter as a header for all state-changing requests. This token is taken from the user's cookie and then passed in the request as part of the Girder one-page application and other clients such that the cookie alone is not enough to form a valid request. A sensible CORS policy (discussed above) also helps mitigate this attack vector.

Dependent Libraries

Another common attack vector is through libraries upon which Girder depends such as Cherrypy, Pug, PyMongo, etc. Girder's library dependencies reference specific versions, ensuring that arbitrary upstream changes to libraries are not automatically accepted into Girder's environment. Conversely, during development and before releases we work to ensure our dependencies are up to date in order to get the latest security fixes.

Notes on Secure Deployment

It is recommended that Girder be deployed using HTTPS as the only access method. Additionally, we recommend encrypting the volume where the Mongo database is stored as well as always connecting to Mongo using authenticated access. The volume containing any on-disk assetstores should also be encrypted to provide encryption of data at rest. We also recommend using a tool such as logrotate to enable the audit of Girder logs in the event of a data breach. Finally, we recommend a regular (and regularly tested) backup of the Girder database, configuration, and assetstores. Disaster recovery is an important part of any security plan.

3.3.8 Build the Sphinx Documentation

In order to build the Sphinx documentation, you can use the Grunt task at the top level like so:

```
grunt docs
```

or manually run the Makefile here:

```
make html
```

This assumes that the Sphinx package is installed in your site packages or virtual environment. If that is not yet installed, it can be done using **pip**.

```
pip install sphinx
```

3.3.9 Migration Guide

This document is meant to guide Girder plugin developers in transitioning between major versions of Girder. Major version bumps contain breaking changes to the Girder core library, which are enumerated in this guide along with instructions on how to update your plugin code to work in the newer version.

1.x → 2.x

Server changes

- The deprecated event `'assetstore.adapter.get'` has been removed. Plugins using this event to register their own assetstore implementations should instead just call the `girder.utility.assetstore_utilities.setAssetstoreAdapter` method at load time.
- The `'model.upload.assetstore'` event no longer supports passing back the target assetstore by adding it to the `event.info` dictionary. Instead, handlers of this event should use `event.addResponse` with the target assetstore as the response.
- The unused `user` parameter of the `updateSize` methods in the collection, user, item, and folder models has been removed.
- The unused `user` parameter of the `isOrphan` methods in the file, item, and folder models has been removed.
- Several core models supported an older, nonstandard kwarg format in their `filter` method. This is no longer supported; the argument representing the document to filter is now always called `doc` rather than using the model name for the kwarg. If you were using positional args or using the `filterModel` decorator, this change will not affect your code.
- Multiple configurable plugin loading paths are no longer supported. Use `girder-install plugin <your_plugin_path>` to install plugins that are not already in the plugins directory. Pass `-s` to that command to symlink instead of copying the directory. This also means:
  - The `plugins.plugin_directory` and `plugins.plugin_install_path` config file settings are no longer supported, but their presence will not cause problems.
  - The `defaultPluginDir`, `getPluginDirs`, `getPluginParentDir` methods inside `girder.utility.plugin_utilities` were removed.
  - All of the methods in `girder.utility.plugin_utilities` no longer accept a `curConfig` argument since the configuration is no longer read.
- The `girder.utility.sha512_state` module has been removed. All of its symbols had been deprecated and replaced by corresponding ones in `girder.utility.hash_state`.

Web client changes

- In version 1.x, running `npm install` would install our npm dependencies, as well as run the web client build process afterwords. That is no longer the case; `npm install` now only installs the dependencies, and the build is run with `npm run build`.
  - The old web client build process used to build *all available* plugins in the plugin directory. Now, running `npm run build` will *only build the core code*. You can pass a set of plugins to additionally build by passing them on the command like, e.g. `npm run build -- --plugins=x,y,z`.
  - The `grunt watch` command has been deprecated in favor of `npm run watch`. This also only watches the core code by default, and if you wish to also include other plugins, you should pass them in the same way, e.g. `npm run watch -- --plugins=x,y,z`.

  - The `girder-install web` command is now the recommended way to build web client code. It builds all *enabled* plugins in addition to the core code. The ability to rebuild the web client code for the core and all enabled plugins has been exposed via the REST API and the admin console of the core web client. The recommended process for administrators is to turn on all desired plugins via the switches, click the **Rebuild web code** button, and once that finishes, click the button to restart the server.

- **Jade → Pug** rename: Due to trademark issues, our upstream HTML templating engine was renamed from jade to pug. In addition, this rename coincides with a major version bump in the language which comes with notable breaking changes.
  - Template files should now end in `.pug` instead of `.jade`. This affects how they are imported as modules in webpack.
  - Jade-syntax interpolation no longer works inside string values of attributes. Use ES2015-style string templating instead. Examples:
    * `a(href="#item/#{id}/foo")` → `a(href=`#item/${id}/foo`)`
    * `.g-some-element(cid="#{obj.cid}")` → `.g-some-element(cid=obj.cid)`
  - Full list of breaking changes are listed here, though most of the others are relatively obscure.
- Testing specs no longer need to manually import all of the source JS files under test. We now have better source mapping in our testing infrastructure, so it's only necessary to import the built target for your plugin, e.g.
  - 1.x:

```
girderTest.addCoveredScripts([
    '/static/built/plugins/jobs/templates.js',
    '/plugins/jobs/web_client/js/misc.js',
    '/plugins/jobs/web_client/js/views/JobDetailsWidget.js',
    '/plugins/jobs/web_client/js/views/JobListWidget.js'
]);
```

  - 2.x:

```
girderTest.addCoveredScripts([
    '/clients/web/static/built/plugins/jobs/plugin.min.js'
]);
```

- **Build system overhaul**: Girder web client code is now built with Webpack instead of uglify, and we use the Babel loader to enable ES2015 language support. The most important result of this change is that plugins can now build their own targets based on the Girder core library in a modular way, by importing specific components. See the *plugin development guide* for a comprehensive guide on developing web-client plugins in the new infrastructure.

Python client changes

- Girder CLI: Subcommands are no longer specified with the `-c` option. Instead, the subcommand is specified just after all the general flags used for connection and authentication. For example:
  - Before: `girder-cli --api-key=abcdefg --api-url=https://mygirder.org/api/v1 -c upload 1234567890abcdef ./foo`
  - After: `girder-cli --api-key=abcdefg --api-url=https://mygirder.org/api/v1 upload 1234567890abcdef ./foo`
- The `blacklist` and `dryrun` kwargs are no longer available in the `GirderClient` constructor because they only apply to uploading. If you require the use of a blacklist, you should now pass it into the `upload` method.

These options can still be passed on the CLI, though they should now come *after* the `upload` subcommand argument.

- Legacy method names in the `GirderClient` class API have been changed to keep naming convention consistent.
  - `add_folder_upload_callback` → `addFolderUploadCallback`
  - `add_item_upload_callback` → `addItemUploadCallback`
  - `load_or_create_folder` → `loadOrCreateFolder`
  - `load_or_create_item` → `loadOrCreateItem`
- All kwargs to `GirderClient` methods have been changed from **snake_case** to **camelCase** for consistency.
- Listing methods in the `GirderClient` class (e.g. `listItem`) have been altered to be generators rather than return lists. By default, they will now iterate until exhaustion, and callers won't have to pass `limit` and `offset` parameters unless they want a specific slice of the results. As long as you are just iterating over results, this will not break your existing code, but if you were using other operations only available on lists, this could break. The recommended course of action is to modify your logic so that you only require iteration over the results, though it is possible to simply wrap the return value in a `list()` constructor. Use caution if you use the `list()` method, as it will load the entire result set into memory.

Built-in plugin changes

- **Jobs**: The deprecated `jobs.filter` event was removed. Use the standard `exposeFields` and `hideFields` methods on the job model instead.
- **OAuth**: For legacy backward compatibility, the Google provider was previously enabled by default. This is no longer the case.

3.4 Plugins

One of the most useful aspects of the Girder platform is its ability to be extended in almost any way by custom plugins. Developers looking for information on writing their own plugins should see the *Plugin Development* section. Below is a listing and brief documentation of some of Girder's standard plugins that come pre-packaged with the application.

3.4.1 Jobs

The jobs plugin is useful for representing long-running (usually asynchronous) jobs in the Girder data model. Since the notion of tracking batch jobs is so common to many applications of Girder, this plugin is very generic and is meant to be an upstream dependency of more specialized plugins that actually create and execute the batch jobs.

The job resource that is the primary data type exposed by this plugin has many common and useful fields, including:

- `title`: The name that will be displayed in the job management console.
- `type`: The type identifier for the job, used by downstream plugins opaquely.
- `args`: Ordered arguments of the job (a list).
- `kwargs`: Keyword arguments of the job (a dictionary).
- `created`: Timestamp when the job was created
- `progress`: Progress information about the job's execution.

- `status`: The state of the job, e.g. Inactive, Running, Success.
- `log`: Log output from this job's execution.
- `handler`: An opaque value used by downstream plugins to identify what should handle this job.
- `meta`: Any additional information about the job should be stored here by downstream plugins.

Jobs should be created with the `createJob` method of the job model. Downstream plugins that are in charge of actually scheduling a job for execution should then call `scheduleJob`, which triggers the `jobs.schedule` event with the job document as the event info.

The jobs plugin contains several built-in status codes within the `girder.plugins.jobs.constants.JobStatus` namespace. These codes represent various states a job can be in, which are:

- INACTIVE (0)
- QUEUED (1)
- RUNNING (2)
- SUCCESS (3)
- ERROR (4)
- CANCELED (5)

Downstream plugins that wish to expose their own custom job statuses must hook into the `jobs.status.validate` event for any new valid status value, which by convention must be integer values. To validate a status code, the default must be prevented on the event, and the handler must add a `True` response to the event. For example, a downstream plugin with a custom job status with the value *1234* would add the following hook:

```
from girder import events

def validateJobStatus(event):
    if event.info == 1234:
        event.preventDefault().addResponse(True)

def load(info):
    events.bind('jobs.status.validate', 'my_plugin', validateJobStatus)
```

Downstream plugins that want to hook into job updates must use a different convention than normal; for the sake of optimizing data transfer, job updates do not occur using the normal `save` method of Girder models. Therefore, plugins that want to listen to job updates should bind to either `jobs.job.update` (which is triggered prior to persisting the updates and can be used to prevent the update) or `jobs.job.update.after` (which is triggered after the update). Users of these events should be aware that the `log` field of the job will not necessarily be in sync with the persisted version, so if your event handler requires access to the job log, you should manually re-fetch the full document in the handler.

3.4.2 Geospatial

The geospatial plugin enables the storage and querying of GeoJSON formatted geospatial data. It uses the underlying MongoDB support of geospatial indexes and query operators to create an API for the querying of items that either intersect a GeoJSON point, line, or polygon; are in proximity to a GeoJSON point; or are entirely within a GeoJSON polygon or circular region. In addition, new items may be created from GeoJSON features or feature collections. GeoJSON properties of the features are added to the created items as metadata.

The plugin requires the geojson Python package, which may be installed using **pip**:

```
pip install -e .[geospatial]
```

Once the package is installed, the plugin may be enabled via the admin console.

3.4.3 Google Analytics

The Google Analytics plugin enables the use of Google Analytics to track page views with the Girder one-page application. It is primarily a client-side plugin with the tracking ID stored in the database. Each routing change will trigger a page view event and the hierarchy widget has special handling (though it does not technically trigger routing events for hierarchy navigation).

To use this plugin, simply copy your tracking ID from Google Analytics into the plugin configuration page.

3.4.4 Homepage

The Homepage plugin allows the default Girder front page to be replaced by content written in [Markdown](https://daringfireball.net/projects/markdown/) format. After enabling this plugin, visit the plugin configuration page to edit and preview the Markdown.

3.4.5 Auto Join

The Auto Join plugin allows you to define rules to automatically assign new users to groups based on their email address domain. Typically, this is used in conjunction with email verification.

When a new user registers, each auto join rule is checked to see if the user's email address contains the rule pattern as a substring (case insensitive).

If there is a match, the user is added to the group with the specified access level.

3.4.6 Metadata Extractor

The metadata extractor plugin enables the extraction of metadata from uploaded files such as archives, images, and videos. It may be used as either a server-side plugin that extracts metadata on the server when a file is added to a filesystem asset store local to the server or as a remote client that extracts metadata from a file on a filesystem local to the client that is then sent to the server using the Girder Python client.

The server-side plugin requires several Hachoir Python packages to parse files and extract metadata from them. These packages may be installed using **pip** as follows:

```
pip install -e .[metadata_extractor]
```

Once the packages are installed, the plugin may be enabled via the admin console on the server.

In this example, we use the girder python client to interact with the plugin's python API. Assuming `girder_client.py` and `metadata_extractor.py` are located in the module path, the following code fragment will extract metadata from a file located at `path` on the remote filesystem that has been uploaded to `itemId` on the server:

```
from girder_client import GirderClient
from metadata_extractor import ClientMetadataExtractor

client = GirderClient(host='localhost', port=8080)
client.authenticate(login, password)
```

```
extractor = ClientMetadataExtractor(client, path, itemId)
extractor.extractMetadata()
```

The user authenticating with `login` and `password` must have `WRITE` access to the file located at `itemId` on the server.

3.4.7 OAuth Login

This plugin allows users to log in using OAuth against a set of supported providers, rather than storing their credentials in the Girder instance. Specific instructions for each provider can be found below.

Google

On the plugin configuration page, you must enter a **Client ID** and **Client secret**. Those values can be created in the Google Developer Console, in the **APIS & AUTH > Credentials** section. When you create a new Client ID, you must enter the `AUTHORIZED_JAVASCRIPT_ORIGINS` and `AUTHORIZED_REDIRECT_URI` fields. These *must* point back to your Girder instance. For example, if your Girder instance is hosted at `https://my.girder.com`, then you should specify the following values:

```
AUTHORIZED_JAVASCRIPT_ORIGINS: https://my.girder.com
AUTHORIZED_REDIRECT_URI: https://my.girder.com/api/v1/oauth/google/callback
```

After successfully creating the Client ID, copy and paste the client ID and client secret values into the plugin's configuration page, and hit **Save**. Users should then be able to log in with their Google account when they click the log in page and select the option to log in with Google.

3.4.8 Curation

This plugin adds curation functionality to Girder, allowing content to be assembled and approved prior to publication. Admin users can activate curation for any folder, and users who are then granted permission can compose content under that folder. The users can request publication of the content when it is ready, which admins may approve or reject. The plugin provides a UI along with workflow management, notification, and permission support for these actions.

The standard curation workflow works as follows, with any operations affecting privacy or permissions being applied to the folder and all of its descendent folders.

- Site admins can enable curation for any folder, which changes the folder to Private.
- Users with write access can populate the folder with data.
- When ready, a user can request approval from the admin. The folder becomes read-only at this point for any user or group with write access, to avoid further changes being made while the admin is reviewing.
- The admin can approve or reject the folder contents.
- If approved, the folder becomes Public.
- If rejected, the folder becomes writeable again by any user or group with read access, enabling users to make changes and resubmit for approval.

The curation dialog is accessible from the Folder actions menu and shows the following information.

- Whether curation is enabled or disabled for the folder.
- The current curation status: construction, requested, or approved.
- A timeline of status changes, who performed them and when.

- Context-dependent action buttons to perform state transitions.

3.4.9 Provenance Tracker

The provenance tracker plugin logs changes to items and to any other resources that have been configured in the plugin settings. Each change record includes a version number, the old and new values of any changed information, the ID of the user that made the change, the current date and time, and the type of change that occurred.

API

Each resource that has provenance tracking has a rest endpoint of the form `(resource)/{id}/provenance`. For instance, item metadata is accessible at `item/{id}/provenance`. Without any other parameter, the most recent change is reported.

The `version` parameter can be used to get any or all provenance information for a resource. Every provenance record has a version number. For each resource, these versions start at 1. If a positive number is specified for `version`, the provenance record with the matching version is returned. If a negative number is specified, the index is relative to the end of the list of provenance records. That is, -1 is the most recent change, -2 the second most recent, etc. A `version` of `all` returns a list of all provenance records for the resource.

All provenance records include `version`, `eventType` (see below), and `eventTime`. If the user who authorized the action is known, their ID is stored in `eventUser`.

Provenance event types include:

- `creation`: the resource was created.
- `unknownHistory`: the resource was created when the provenance plugin was disabled. Prior to this time, there is no provenance information.
- `update`: data, metadata, or plugin-related data has changed for the resource. The old values and new values of the data are recorded. The `old` parameter contains any value that was changed (the value prior to the change) or has been deleted. The `new` parameter contains any value that was changed or has been added.
- `copy`: the resource was copied. The original resource's provenance is copied to the new record, and the `originalId` indicates which record was used.

For item records, when a file belonging to that item is added, removed, or updated, the provenance is updated with that change. This provenance includes a `file` list with the changed file(s). Each entry in this list includes a `fileId` for the associated file and one of these event types:

- `fileAdded`: a file was added to the item. The `new` parameter has a summary of the file information, including its assetstore ID and value used to reference it within that assetstore.
- `fileUpdate`: a file's name or other data has changed, or the contents of the file were replaced. The `new` and `old` parameters contain the data values that were modified, deleted, or added.
- `fileRemoved`: a file was removed from the item. The `old` parameter has a summary of the file information. If this was the only item using this file data, the file is removed from the assetstore.

3.4.10 Gravatar Portraits

This lightweight plugin makes all users' Gravatar image URLs available for use in clients. When enabled, user documents sent through the REST API will contain a new field `gravatar_baseUrl` if the value has been computed. If that field is not set on the user document, instead use the URL `/user/:id/gravatar` under the Girder API, which will compute and store the correct Gravatar URL, and then redirect to it. The next time that user document is sent over the REST API, it should contain the computed `gravatar_baseUrl` field.

Javascript clients

The Gravatar plugin's javascript code extends the Girder web client's `girder.models.UserModel` by adding the `getGravatarUrl(size)` method that adheres to the above behavior internally. You can use it on any user model with the `_id` field set, as in the following example:

```
import { getCurrentUser } from 'girder/auth';

const currentUser = getCurrentUser();
if (currentUser) {
    this.$('div.gravatar-portrait').css(
        'background-image', `url(${currentUser.getGravatarUrl(36)})`);
}
```

Note: Gravatar images are always square; the `size` parameter refers to the side length of the desired image in pixels.

3.4.11 HDFS Assetstore

This plugin creates a new type of assetstore that can be used to store and proxy data on a Hadoop Distributed Filesystem. An HDFS assetstore can be used to import existing HDFS data hierarchies into the Girder data hierarchy, and it can also serve as a normal assetstore that stores and manages files created via Girder's interface.

Note: Deleting files that were imported from existing HDFS files does not delete the original file from HDFS, they will simply be unlinked in the Girder hierarchy.

Once you enable the plugin, site administrators will be able to create and edit HDFS assetstores on the `Assetstores` page in the web client in the same way as any other assetstore type. When creating or editing an assetstore, validation is performed to ensure that the HDFS instance is reachable for communication, and that the directory specified as the root path exists. If it does not exist, Girder will attempt to create it.

Importing data

Once you have created an HDFS assetstore, you will be able to import data into it on demand if you have site administrator privileges. In the assetstore list in the web client, you will see an **Import** button next to your HDFS assetstores that will allow you to import files or directories (recursively) from that HDFS instance into a Girder user, collection, or folder of your choice.

You should specify an absolute data path when importing; the root path that you chose for your assetstore is not used in the import process. Each directory imported will become a folder in Girder, and each file will become an item with a single file inside. Once imported, file data is proxied through Girder when being downloaded, but still must reside in the same location on HDFS.

Duplicates (that is, pre-existing files with the same name in the same location in the Girder hierarchy) will be ignored if, for instance, you import the same hierarchy into the same location twice in a row.

3.4.12 Remote Worker

This plugin should be enabled if you want to use the Girder worker distributed processing engine to execute batch jobs initiated by the server. This is useful for deploying service architectures that involve both data management and scalable offline processing. This plugin provides utilities for sending generic tasks to worker nodes for execution. The

worker itself uses celery to manage the distribution of tasks, and builds in some useful Girder integrations on top of celery. Namely,

- **Data management**: This plugin provides python functions for building task input and output specs that refer to data stored on the Girder server, making it easy to run processing on specific folders, items, or files. The worker itself knows how to authenticate and download data from the server, and upload results back to it.
- **Job management**: This plugin depends on the *Jobs plugin*. Tasks are specified as python dictionaries inside of a job document and then scheduled via celery. The worker automatically updates the status of jobs as they are received and executed so that they can be monitored via the jobs UI in real time. If the script prints any logging information, it is automatically collected in the job log on the server, and if the script raises an exception, the job status is automatically set to an error state.

CHAPTER 4

API index

- genindex
- modindex

g

Symbols

A

B

C

D

H

I

J

L

M

N

P

R

S

T

U

V