Girder Documentation

Release 1.3.1

Kitware

May 14, 2015

Contents

CHAPTER 1

What is Girder?

Girder is a free and open source web-based **data management platform**. 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

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
```

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
```

Enable the Node.js APT repository:

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

Install Node.js and NPM using APT:

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

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
```

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 | 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 remove the line for bcrypt from the `requirements.txt` file and manually install it. 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.

Install with pip

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

```
pip install girder
```

This will install the core Girder REST API 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 download and install these libraries for you. Just run the following command:

```
girder-install web
```

If you installed Girder into your system `site-packages`, you may need to run this command as root.

Optionally, you can also install a set of *plugins* that are distributed with Girder. The `girder-install` script can do this for you as well by running:

```
girder-install plugin
```

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

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 -r requirements.txt
```

Note: If you intend to develop Girder or want to run the test suite, you should also install the development dependencies:

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

One of the development requirements, *httpretty*, can fail to install if under certain system locales (see the error report). Changing to any UTF8 locale works around this problem. For instance, on Ubuntu, you can change your system locale using the commands:

```
locale-gen en_US.UTF-8
export LANG=en_US.utf8
```

Before you can build the client-side code project, you must install the Grunt command line utilities:

```
npm install -g grunt-cli
```

Then cd into the root of the repository and run:

```
npm install
```

This should run `grunt init` and `grunt` 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 &
```

If you installed with pip, you will have the `girder-server` executable on your path and can simply call

```
girder-server
```

- or -

If you checked out the source tree, you can start the server with the following command, which will have identical behavior:

```
python -m girder
```

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`.

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 X-Forwarded-Host $http_host;
    proxy_set_header X-Forwarded-Server $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    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;
}
```

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

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

The `tools.proxy.base` and `tools.proxy.local` aren't necessary if the proxy service adds the appropriate X-Forwarded-Host header to proxied requests. For this purpose, the X-Forwarded-Host should be the host used in http requests, including any non-default port.

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

```
$ grunt init && grunt
```

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 databse, 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 it's 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 ouput 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.

Plugin path

When checking out Girder from source (recommended), the plugin directory will be set to the *plugins* directory by default. If Girder is installed from PyPi (experimental), then the plugin directory can be set in the *plugin_directory* of the *plugins* section.

Server thread pool

Girder can handle mutiple 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 priveleges, 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.

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 or email us.

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.

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

Model Base

Events

User

Password

Token

Group

Collection

Folder

Item

Setting

Python API for RESTful web API

Base Classes and Helpers

User

Group

Item

Folder

Utility

Constants

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 Bootsrap 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 can refer to the System Prerequisites and Installation sections to setup a local development environment. Once Girder is started via `python -m girder`, the server will reload itself whenever a Python file is modified.

To get the same auto-building behavior for JavaScript, we use `grunt-watch`. Thus, running `grunt watch` in the root of the repository will watch for JavaScript, Stylus, and Jade changes in order to rebuild them on-the-fly. If you do not run `grunt watch` while making code changes, you will need to run the `grunt` 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.

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

Creating a new release

Girder releases are uploaded to PyPI for easy installation via `pip`. In addition, the python source package and optional plugin and web client 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 `docs/conf.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 three new tarballs in the current directory:

   girder-<version>.tar.gz This is the python source distribution for the core server API.

   girder-web-<version>.tar.gz This is the web client libraries.

   girder-plugins-<version>.tar.gz This contains all of the plugins in the main repository.

5. Create a new virtual environment and install the python package into it as well as the optional web and plugin components. 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 -s ../girder-web-<version>.tar.gz
girder-install plugin -s ../girder-plugins-<version>.tar.gz
```

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 three tarballs 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/setup.py` according to the above rules. It is set in the line near the top of the file that looks like `CLIENT_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:

```
{
"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 as in the following example:

```
from girder.api.describe import Description
from girder.api import access

@access.public
def myHandler(id, params):
    return {
       'itemId': id,
       'cat': params.get('cat', 'No cat param passed')
    }
myHandler.description = (
    Description('Retrieve the cat for a given item.')
    .param('id', 'The item ID', paramType='path')
    .param('cat', 'The cat value.', required=False)
    .errorResponse())
```

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 set the handler's description to `None`, and then no warning will appear.

```
myHandler.description = 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):
        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`.

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.

Extending the Client-Side Application

The web client may be extended independently of the server side. Plugins may import Jade 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, there are three optional subdirectories that can be used to import content:

- `stylesheets`: Any files ending with **.styl** in this directory or any of its subdirectories will be automatically built into CSS and loaded if your plugin is enabled. These files must obey Stylus syntax. Because these CSS scripts are imported *after* all of the core CSS, any rules you write will override any existing core style rules.
- `templates`: Any files ending with **.jade** in this directory or any of its subdirectories will be automatically built as templates available in the application. Just like in core, these templates are uniquely identified by the name of their file; e.g., `myTemplate.jade` could be rendered at runtime by calling `girder.templates.myTemplate()`. So, if you want to override an existing core template, simply create one in this directory with the same name. If you want to create a template that is not an override of a core template, but simply belongs to your plugin, convention dictates that it should begin with your plugin name followed by an underscore to avoid collisions, e.g., `cats_catPage.jade`. Documentation for the Jade language can be found here.
- `js`: Any files ending with **.js** in this directory or any of its subdirectories will be compiled using uglify and imported into the front-end application. The compiled JavaScript file will be loaded after all of the core JavaScript files are loaded, so it can access all of the objects declared by core. The source map for these files will be automatically built and served as well.
- `extra`: Any files in this directory or any of its subdirectories will be copied into the **extra** directory under your plugin's built static directory. Any additional public static content that is required by your plugin that doesn't fall into one of the above categories can be placed here, such as static images, fonts, or third-party static libraries.

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" ]
    }
}
```

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.

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 the `girder.events` object. 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 the `girder.wrap` function to *wrap* the method of the core prototype with our own function.

```
girder.wrap(girder.views.HierarchyWidget, 'render', function (render) {
    // Call the underlying render function that we are wrapping
    render.call(this);

    // Add a link just below the widget
    this.$('.g-hierarchy-widget').after('<a class="cat-link">Meow</a>');
});
```

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:

```
girder.views.HierarchyWidget.prototype.events['click a.cat-link'] = function () {
    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.

```
girder.router.route('collection/:id/frontPage', 'collectionFrontPage', function (collectionId, params
    var collection = new girder.models.CollectionModel();
    collection.set({
        _id: collectionId
    }).on('g:fetched', function () {
        girder.events.trigger('g:navigateTo', girder.views.CollectionView, _.extend({
            collection: collection
        }, params || {}), {layout: girder.Layout.EMPTY});
    }, this).on('g:error', function () {
        girder.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 `Authorization` header:

```
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.

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.

```
var fileModel = new girder.models.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.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:

```
new girder.views.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/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. A common example would be downloading a file from the server; we want to send just the data, not try to encode it in JSON.

If you want to send a raw response, simply make your route handler return a generator function. In Girder, a raw response is also automatically a streaming response, giving developers full control of the buffer size of the response body. That is, 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 send 10 chunks to the client, and the full response body would be `0123456789`.

```
@access.public
def rawExample(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)
```

Serving a custom app from the server root

Normally, the root node (`/`) of the server will serve up the Girder web client. Some plugins will wish to change this so that their own custom app gets served out of the server root instead, and they may also want to move the Girder web client to be served out of an alternative route so they can still use it in addition to their custom front-end application.

To achieve this, you simply have to swap the existing server root with your own and rebind the old app underneath. In your plugin's `load` method, you would add something like the following:

```
info['serverRoot'], info['serverRoot'].girder = CustomAppRoot(), info['serverRoot']
```

This will make it so that `/` serves your `CustomAppRoot`, and `/girder` will serve the normal Girder web client. That also has the side effect of moving the web API (normally `/api`) as well; it would now be moved to `/girder/api`, which would require a change to the `server.api_root` value in `girder.local.cfg`.

If you would rather your web API remained at `/api` instead of moving under `/girder/api`, you would simply have to move it underneath the new server root. To do that, just add the following line below the previous line:

```
info['serverRoot'].api = info['serverRoot'].girder.api
```

This will now serve the api out of *both* `/api` and `/girder/api`, which may be desirable. If you only want it to be served out of `/api` and not `/girder/api`, just add a final line below that:

```
del info['serverRoot'].girder.api
```

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. You should only do this if the endpoint is "read-only", that is, in cases where it does not make modifications to data on the server, to avoid vulnerabilities to Cross-Site Request Forgery attacks. 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 as a last resort, but in practice this will probably never be the case.

In order to allow cookie authentication for your route, simply set the `cookieAuth` property on your route handler function to `True`. Example:

```
@access.public
def download(self, params):
    ...
download.cookieAuth = True
```

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/libs.min.js"></script>
<script src="/girder/static/built/app.min.js"></script>
```

Note that `libs.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 `libs.min.js`.

Initializing Girder

The following code will initialize the Girder environment and should be set before performing any Girder API calls:

```
$(document).ready(function () {
    girder.apiRoot = '/girder/api/v1';
    girder.router.enabled(false);

    // Your app code here
});
```

Note that `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 user Girder UI components, you will need Bootstrap and Girder CSS in your HTML:

```
<link rel="stylesheet" href="/girder/static/lib/bootstrap/css/bootstrap.min.css">
<link rel="stylesheet" href="/girder/static/built/app.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 <arguments>
```

Specifying the Girder Instance

The default host for the girder_client is *localhost*. To specify the host with cli usage

```
python cli.py --host girder.example.com
```

To specify a host using SSL (https)

```
python cli.py --host girder.example.com --scheme https --port 443
```

Upload a local file hierarchy

The girder_client can upload to an S3 Assetstore when uploading to a Girder server that is version 1.3.0 or later.

The upload command, `-c upload`, is the default, so the following two forms are equivalent

```
python cli.py
python cli.py -c upload
```

To upload a folder hierarchy rooted at *test_folder* to the Girder Folder with id *54b6d41a8926486c0cbca367*

```
python cli.py 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

```
python cli.py 54b6d41a8926486c0cbca367 test_folder
python cli.py 54b6d41a8926486c0cbca367 test_folder --parent-type folder
```

To upload that same local folder to a Collection or User, specify the parent type as follows

```
python cli.py 54b6d41a8926486c0cbca459 test_folder --parent-type user
```

To see what local folders and files on disk would be uploaded without actually uploading anything, add the `--dryrun` flag

```
python cli.py 54b6d41a8926486c0cbca367 test_folder --dryrun
```

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

```
python cli.py 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.

```
python cli.py 54b6d41a8926486c0cbca367 test_folder --reuse
```

To include a blacklist of filepatterns that will not be uploaded, pass a comma separated list to the `--blacklist` arg

```
python cli.py 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*

```
python cli.py -c download 54b6d40b8926486c0cbca364 download_folder
```

Downloading is only supported from a parent type of Folder.

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()
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 folder_callback(folder, filepath):
    # assume we have a folder_metadata dict that has
    # filepath: metadata_dict_for_folder
    gc.addMetadataToFolder(folder['_id'], folder_metadata[filepath])

def item_callback(item, filepath):
    # assume we have an item_metadata dict that has
    # filepath: metadata_dict_for_item
    gc.addMetadataToItem(item['_id'], item_metadata[filepath])

gc.authenticate('username', 'password')
gc.add_folder_upload_callback(folder_callback)
gc.add_item_upload_callback(item_callback)
gc.upload(local_folder, parent_id)
```

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 (Jade). 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, Jade, 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
```

3.4 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.

For controlling what fields of a job are visible in the REST API, downstream plugins should bind to the `jobs.filter` event, which receives a dictionary with `job` and `user` keys as its info. They can modify any existing fields or the job document as needed, and can also expose or redact fields. To make some fields visible while redacting others, you can use the event response with `exposeFields` and/or `removeFields` keys, e.g.

```
def filterJob(event):
    event.addResponse({
        'exposeFields': ['_some_other_field'],
        'removeFields': ['created']
    })

events.bind('jobs.filter', 'a_downstream_plugin', filterJob)
```

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 -r plugins/geospatial/requirements.txt
```

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 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 -r plugins/metadata_extractor/requirements.txt
```

Once the packages are installed, the plugin may be enabled via the admin console on the server.

The remote client requires the same Python packages as the server plugin, but additionally requires the Requests Python package to communicate with the server using the Girder Python client. These packages may be installed using **pip** as follows:

```
pip install requests -r plugins/metadata_extractor/requirements.txt
```

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.

3.4.5 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.

3.4.6 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.

3.4.7 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:

```
if (girder.currentUser) {
    this.$('div.gravatar-portrait').css(
        'background-image', 'url(' +
        girder.currentUser.getGravatarUrl(36) + ')');
}
```

Note: Gravatar images are always square; the `size` parameter refers to the side length of the desired image in pixels.

3.4.8 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 communciation, 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.

CHAPTER 4

API index

- genindex
- modindex

Symbols