Girder Documentation

Release 1.0.1

Kitware

October 27, 2014

Contents

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

CHAPTER 1

Administrator Documentation

1.1 System Prerequisites

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

- Python 2
- 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.

- *Debian / Ubuntu*
- *CentOS / Fedora / Red Hat Enterprise Linux*
- *OS X*
- *Windows*

1.1.1 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
```

1.1.2 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
```

1.1.3 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.

1.1.4 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
```

1.2 Installation

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

1.2.1 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*.

1.2.2 Install from Git Checkout

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

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

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
```

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
```

Finally, when all Node packages are installed, run:

```
grunt init
```

To build the client-side code, run the following command from within the repository:

```
grunt
```

Run this command any time you change a JavaScript or CSS file under __*clients/web*__.

1.2.3 Run

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

```
mongod &
```

Then, just run:

```
python -m girder
```

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

1.2.4 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`.

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.

1.3.1 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!

1.3.2 Apache Reverse Proxy

In many cases, it is useful to route multiple web services from a single server. You can configure Apache's mod_proxy to route traffic to these services using a reverse proxy. For example, if you have an Apache server accepting requests at `www.example.com`, and you want to forward requests to `www.example.com/girder` to a Girder instance listening on port `9000`. You can add the following section to your Apache config:

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

In such a scenario, Girder must be configured properly in order to serve content correctly. Fortunately, 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"
```

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

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

1.3.3 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
```

CHAPTER 2

User Documentation

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).

Our aims for Girder is for it to be robust, performant, extensible, and grokable.

Girder is built in Python.

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

2.1.1 Document Conventions

This document is written for end-users of Girder, rather than developers. Since it was written by developers, sometimes we fail at making this distinction, please remind (and forgive) us.

Girder specific entities will be `formatted like this`.

2.1.2 Concepts

Users

This is a common software concept, isn't it nice that we didn't change its established meaning! Each user of Girder should have a `User` created within Girder. Their Girder `User` will determine their permissions and can store and share their data.

Groups

`Groups` group together `Users`; the most common usage example would be to give access to specific resources to any member of a `Group`.

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 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>
```

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.

Collections

A Girder Collection is functional top level grouping of Folders. A Collection collects resources (Folders, Items, and Users) that should have some common usage, e.g., for a particular project.

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.

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 metadata about the resource)
4. ADMIN permission (includes READ and WRITE permission, can delete the resource)

A site admin 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`.

CHAPTER 3

Developer Documentation

3.1 API Documentation

3.1.1 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.

3.1.2 Models

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

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

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

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

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

```
from girder.models.model_base import Model

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

The above model singleton could then be accessed via:

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

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

Model Helper Functions

`girder.models.`**`getDbConfig`**`()`

Get the database configuration values from the cherrypy config.

`girder.models.`**`getDbConnection`**`()`

Get a MongoClient object that is connected to the configured database. We lazy-instantiate a module-level singleton, the MongoClient objects manage their own connection pools internally.

Model Base

class `girder.models.model_base.`**`AccessControlledModel`**

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

`copyAccessPolicies`(*src*, *dest*, *save=False*)

Copies the set of access control policies from one document to another.

Parameters

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

Returns The modified destination document.

`filterResultsByPermission`(*cursor*, *user*, *level*, *limit*, *offset*, *removeKeys=()*)

Given a database result cursor, this generator will yield only the results that the user has the given level of access on, respecting the limit and offset specified.

Parameters

- **cursor** – The database cursor object from "find()".
- **user** – The user to check policies against.
- **level** (*AccessType*) – The access level.
- **limit** – The max size of the result set.
- **offset** – The offset into the result set.
- **removeKeys** (*list*) – List of keys that should be removed from each matching document.

`getAccessLevel`(*doc*, *user*)

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

Parameters

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

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

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

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

Parameters

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

Returns Whether the access is granted.

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

Parameters

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

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

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

Parameters

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

Returns The updated resource.

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

Parameters

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

Returns The updated resource document.

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

Parameters

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

Returns The updated resource document.

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

Parameters

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

Returns The modified resource document.

textSearch(*query*, *user=None*, *filters=None*, *limit=50*, *offset=0*, *sort=None*, *fields=None*)
Custom override of Model.textSearch to also force permission-based filtering. The parameters are the same as Model.textSearch, except:

Parameters **user** – The user to apply permission filtering for.

exception `girder.models.model_base.`**AccessException**
Represents denial of access to a resource.

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

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

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

ensureTextIndex(*index*, *language='english'*)
Call this during initialize() of the subclass if you want your model to have a full-text searchable index. Each collection may have zero or one full-text index. :param language: The default_language value for the text index, which is used for stemming and stop words. If the text index should not use stemming and stop words, set this param to 'none'. :type language: str

filterDocument(*doc*, *allow=None*)
This method will filter the given document to make it suitable to output to the user.

Parameters

- **doc** (*dict*) – The document to filter.

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

find(*query=None*, *offset=0*, *limit=50*, ***kwargs*)
Search the collection by a set of parameters. Passes any kwargs through to the underlying pymongo.collection.find function.

Parameters

- **query** (*dict*) – The search query (see general MongoDB docs for "find()")
- **offset** (*int*) – The offset into the results
- **limit** (*int*) – Maximum number of documents to return
- **sort** (*List of (key, order) tuples.*) – The sort order.
- **fields** (*List of strings*) – A mask for filtering result documents by key.

Returns A pymongo database cursor.

findOne(*query=None*, ***kwargs*)
Search the collection by a set of parameters. Passes any kwargs through to the underlying pymongo.collection.find function.

Parameters

- **query** (*dict*) – The search query (see general MongoDB docs for "find()")
- **sort** (*List of (key, order) tuples.*) – The sort order.
- **fields** (*List of strings*) – A mask for filtering result documents by key.

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

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

Parameters

- **query** (*dict*) – The query selector for documents to update.
- **field** (*str*) – The name of the field in the document to increment.
- **amount** (*int or float*) – The amount to increment the field by.

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

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

Parameters

- **id** (*string or ObjectId*) – The value for searching the _id field.
- **objectId** (*bool*) – Whether the id should be coerced to ObjectId type.
- **fields** – Fields list to include. Also can be a dict for exclusion. See pymongo docs for how to use this arg.
- **exc** (*bool*) – Whether to raise a ValidationException if there is no document with the given id.

Returns The matching document, or None.

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

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

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

Parameters

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

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

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

textSearch(*query*, *offset=0*, *limit=50*, *sort=None*, *fields=None*, *filters=None*)
Perform a full-text search against the text index for this collection.

Parameters

- **query** (*str*) – The text query. Will be stemmed internally.
- **filters** (*dict*) – Any additional query operators to apply.

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

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

This is a thin wrapper around pymongo db.collection.update().

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

Parameters

- **query** (*dict*) – The query for finding documents to update. It's the same format as would be passed to find().
- **update** (*dict*) – The update specifier.

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

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

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

Events

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

Listeners should bind to events by calling:

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

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

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

And if the event should be handled asynchronously, use:

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

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

class `girder.events.`**`AsyncEventsThread`**

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

`run``()`

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

`stop``()`

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

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

Adds a new event on the queue to trigger asynchronously.

Parameters

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

class `girder.events.`**`Event`**`(`*name*, *info*`)`

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

`addResponse``(`*response*`)`

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

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

`preventDefault``()`

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

`stopPropagation``()`

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

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

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

Parameters

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

`girder.events.`**`trigger`**(*eventName*, *info=None*)

Fire an event with the given name. All listeners bound on that name will be called until they are exhausted or one of the handlers calls the stopPropagation() method on the event.

Parameters

- **eventName** (*str*) – The name that identifies the event.
- **info** – The info argument to pass to the handler function. The type of this argument is opaque, and can be anything.

`girder.events.`**`unbind`**(*eventName*, *handlerName*)

Removes the binding between the event and the given listener.

Parameters

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

`girder.events.`**`unbindAll`**()

Clears the entire event map. Any bound listeners will be unbound.

User

class `girder.models.user.`**`User`**

This model represents the users of the system.

`createUser`(*login*, *password*, *firstName*, *lastName*, *email*, *admin=False*, *public=True*)

Create a new user with the given information. The user will be created with the default “Public” and “Private” folders.

Parameters

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

Returns The user document that was created.

`fileList`(*doc*, *user=None*, *path=‘’*, *includeMetadata=False*, *subpath=True*)

Generate a list of files within this user’s folders. :param doc: the user to list. :param user: a user used to validate data that is returned. :param path: a path prefix to add to the results. :param includeMetadata: if True and there is any metadata, include a

result which is the json string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.

Parameters **subpath** – if True, add the user's name to the path.

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

Parameters

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

search(*text=None*, *user=None*, *limit=50*, *offset=0*, *sort=None*)
List all users. Since users are access-controlled, this will filter them by access policy.

Parameters

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

Returns List of users.

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

Parameters

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

subtreeCount(*doc*)
Return the size of the user's folders. The user is counted as well.

Parameters **doc** – The user.

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

Password

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

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

Parameters

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

Returns Whether authentication succeeded (bool).

encryptAndStore(*password*)

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

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

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

Token

class girder.models.token.**Token**

This model stores session tokens for user authentication.

createToken(*user=None*, *days=180*)

Creates a new token for the user. You can create an anonymous token (such as for CSRF mitigation) by passing "None" for the user argument.

Parameters

- **user** (*dict*) – The user to create the session for.
- **days** (*int*) – The lifespan of the session in days.

Returns The token document that was created.

girder.models.token.**genToken**(*length=64*)

Use this utility function to generate a random string of a desired length.

Group

class girder.models.group.**Group**

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

Group membership is stored in the database on the user document only; there is no "users" field in this model. This is to optimize for the most common use case for querying membership, which involves checking access control policies, which is always done relative to a specific user. The task of querying all members within a group is much less common and typically only performed on a single group at a time, so doing a find on the indexed group list in the user collection is sufficiently fast.

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

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

addUser(*group*, *user*, *level=0*)

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

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

Parameters

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

Returns The group document that was created.

`filter`(*group*, *user*, *accessList=False*, *requests=False*)
Filter a group document for display to the user.

Parameters

- **group** (*dict*) – The document to filter.
- **user** (*dict*) – The current user.
- **accessList** (*bool*) – Whether to include the access control list field.
- **requests** (*bool*) – Whether to include the requests list field.

Returns The filtered group document.

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

Parameters

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

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

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

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

`getInvites`(*group*, *limit=50*, *offset=0*, *sort=None*)
Return a page of outstanding invitations to a group. This is simply a list of users invited to the group currently.

Parameters

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

`getMembers`(*group*, *offset=0*, *limit=50*, *sort=None*)
Return the list of all users who belong to this group.

Parameters

- **group** – The group to list members on.
- **offset** – Offset into the result set of users.

- **limit** – Result set size limit.
- **sort** – Sort parameter for the find query.

Returns List of user documents.

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

Parameters

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

Returns Whether the access is granted.

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

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

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

`list` (*user=None*, *limit=50*, *offset=0*, *sort=None*)
Search for groups or simply list all visible groups.

Parameters

- **text** – Pass this to perform a text search of all groups.
- **user** – The user to search as.
- **limit** – Result set size limit.
- **offset** – Offset into the results.
- **sort** – The sort direction.

`listMembers` (*group*, *offset=0*, *limit=50*, *sort=None*)
List members of the group, with names, ids, and logins.

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

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

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

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

`updateGroup` (*group*)
Updates a group.

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

Returns The group document that was edited.

Collection

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

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

`createCollection`(*name*, *creator*, *description=''*, *public=True*)

Create a new collection.

Parameters

- **name** (*str*) – The name of the collection. Must be unique.
- **description** (*str*) – Description for the collection.
- **public** (*bool*) – Public read access flag.
- **creator** (*dict*) – The user who is creating this collection.

Returns The collection document that was created.

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

Generate a list of files within this collection's folders. :param doc: the collection to list. :param user: a user used to validate data that is returned. :param path: a path prefix to add to the results. :param includeMetadata: if True and there is any metadata, include a

result which is the json string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.

Parameters **subpath** – if True, add the collection's name to the path.

`filter`(*collection*, *user=None*)

Helper to filter the collection model.

`list`(*user=None*, *limit=50*, *offset=0*, *sort=None*)

Search for collections with full text search.

`remove`(*collection*, *progress=None*, ***kwargs*)

Delete a collection recursively.

Parameters

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

`subtreeCount`(*doc*)

Return the size of the folders within the collection. The collection is counted as well.

Parameters **doc** – The collection.

`updateCollection`(*collection*)

Updates a collection.

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

Returns The collection document that was edited.

Folder

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

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

`childFolders`(*parent*, *parentType*, *user=None*, *limit=50*, *offset=0*, *sort=None*, *filters=None*, ***kwargs*)

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

Parameters

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

`childItems`(*folder*, *limit=50*, *offset=0*, *sort=None*, *filters=None*, ***kwargs*)

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

Parameters

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

`createFolder`(*parent*, *name*, *description='‘*, *parentType='folder'*, *public=None*, *creator=None*)

Create a new folder under the given parent.

Parameters

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

Returns The folder document that was created.

`fileList`(*doc*, *user=None*, *path='‘*, *includeMetadata=False*, *subpath=True*)

Generate a list of files within this folder. :param doc: the folder to list. :param user: the user used for access. :param path: a path prefix to add to the results. :param includeMetadata: if True and there is any metadata, include a result which is the json string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the folder.

Parameters **subpath** – if True, add the folder's name to the path.

`filter` (*folder*, *user*)
Filter a folder document for display to the user.

`getSizeRecursive` (*folder*)
Calculate the total size of the folder by recursing into all of its descendent folders.

`load` (*id*, *level=2*, *user=None*, *objectId=True*, *force=False*, *fields=None*, *exc=False*)
We override load in order to ensure the folder has certain fields within it, and if not, we add them lazily at read time.

Parameters

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

`move` (*folder*, *parent*, *parentType*)
Move the given folder from its current parent to another parent object. Raises an exception if folder is an ancestor of parent.

Parameters

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

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

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

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

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

Parameters

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

`setMetadata` (*folder*, *metadata*)
Set metadata on a folder. A rest exception is thrown in the cases where the metadata json object is badly formed, or if any of the metadata keys contains a period ('.').

Parameters

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

Returns the folder document

subtreeCount (*folder*)
Return the size of the subtree rooted at the given folder. Includes the root folder in the count. Counts folders and items. This returns the absolute size of the subtree, it does not filter by permissions.

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

updateFolder (*folder*)
Updates a folder.

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

Returns The folder document that was edited.

Item

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

childFiles (*item*, *limit=50*, *offset=0*, *sort=None*, ***kwargs*)
Generator function that yields child files in the item. Passes any kwargs to the find function.

Parameters

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

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

Parameters

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

Returns the new item.

createItem (*name*, *creator*, *folder*, *description=''*)
Create a new item. The creator will be given admin access to it.

Parameters

- **name** (*str*) – The name of the item.
- **description** (*str*) – Description for the item.
- **folder** – The parent folder of the item.
- **creator** (*dict*) – User document representing the creator of the group.

Returns The item document that was created.

fileList (*doc*, *user=None*, *path=''*, *includeMetadata=False*, *subpath=True*)
Generate a list of files within this item. :param doc: the item to list. :param user: a user used to validate data that is returned. This isn't

used, but is present to be consistent across all model implementations of fileList.

Parameters

- **path** – a path prefix to add to the results.
- **includeMetadata** – if True and there is any metadata, include a result which is the json string of the metadata. This is given a name of metadata[-(number).json that is distinct from any file within the item.
- **subpath** – if True and the item has more than one file, metadata, or the sole file is not named the same as the item, then the returned paths include the item name.

filter (*item*)
Filter an item document for display to the user.

filterResultsByPermission (*cursor*, *user*, *level*, *limit*, *offset*, *removeKeys=()*)
This method is provided as a convenience for filtering a result cursor of items by permissions, based on the parent folder. The results in the cursor must contain the folderId field.

hasAccess (*item*, *user=None*, *level=0*)
Test access for a given user to this item. Simply calls this method on the parent folder.

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

Parameters

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

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

Parameters

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

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

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

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

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

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

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

Parameters

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

Returns the item document

`textSearch`(*query*, *user=None*, *filters=None*, *limit=50*, *offset=0*, *sort=None*, *fields=None*)
Custom override of Model.textSearch to filter items by permissions of the parent folder.

`updateItem`(*item*)
Updates an item.

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

Returns The item document that was edited.

Setting

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

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

Parameters

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

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

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

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

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

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

Parameters

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

Returns The document representing the saved Setting.

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

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

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

3.1.3 Python API for RESTful web API

Base Classes and Helpers

exception `girder.api.rest.`**`RestException`**(*message*, *code=400*, *extra=None*)

Throw a RestException in the case of any sort of incorrect request (i.e. user/client error). Login and permission failures should set a 403 code; almost all other validation errors should use status 400, which is the default.

`girder.api.rest.`**`endpoint`**(*fun*)

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

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

class `girder.api.rest.`**`loadmodel`**(*map*, *model*, *plugin='_core'*, *level=None*)

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

Parameters

- **map** (*dict*) – Map of incoming parameter name to corresponding model arg name.
- **model** (*str*) – The model name, e.g. 'folder'
- **plugin** (*str*) – Plugin name, if loading a plugin model.
- **level** (*AccessType*) – Access level, if this is an access controlled model.

User

Group

Item

Folder

3.1.4 Utility

class `girder.utility.model_importer.`**`ModelImporter`**

Any class that wants to have convenient model importing semantics should extend/mixin this class.

`model`(*model*, *plugin='_core'*)

Call this to get the instance of the specified model. It will be lazy-instantiated.

Parameters

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

Returns The instantiated model, which is a singleton.

`girder.utility.model_importer.`**`clearModels`**()

Force reloading of all models by clearing the singleton cache. This is used by the test suite to ensure that indices are built properly at startup.

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

Parameters

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

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

Parameters **dir** (*str*) – The directory to add to the template lookup path.

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

Parameters

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

Returns The rendered template as a string of HTML.

`girder.utility.mail_utils.`**`sendEmail`**(*to*, *subject*, *text*)
Send an email. This builds the appropriate email object and then triggers an asynchronous event to send the email (handled in _sendmail).

Parameters

- **to** (*str*) – The recipient's email address.
- **subject** (*str*) – The subject line of the email.
- **text** (*str*) – The body of the email.

3.1.5 Constants

Constants should be defined here.

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

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

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

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

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

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

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

class `girder.constants.`**`TerminalColor`**

Provides a set of values that can be used to color text in the terminal.

3.1.6 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.

3.2 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.

3.2.1 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.

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.

3.2.2 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:

```python
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.

Sending Emails

Girder has utilities that make it easy to send emails. 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`.

Note: All emails are sent as rich text (`text/html` MIME type).

3.2.3 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`.

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.

3.2.4 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.

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**.

Creating 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):

```python
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)
```

3.2.5 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
   ```

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.

3.3.1 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.json** 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.

```
touch cats/plugin.json
```

This JSON 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.",
"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 will be required in order for the change to take effect.

3.3.2 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

@access.public
def myHandler(id, params):
    return {
       'itemId': id,
       'cat': params.get('cat', 'No cat param passed')
    }

def load(info):
    info['apiRoot'].item.route('GET', (':id', 'cat'), myHandler)
```

You should always add an access decorator to your handler function 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).

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.

- **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 or override it completely using `preventDefault` and `addResponse` on the event.

- **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 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.

- **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.

- **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.

3.3.3 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 `jade.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.

3.4 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.

3.4.1 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.

3.4.2 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.handleRouting = false;

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

Note that `girder.handleRouting` 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:

```
$('#login').click(function () {
    var loginView = new girder.views.LoginView({
        el: this.$('#dialog-container')
    });
    loginView.render();
});

$('#register').click(function () {
    var registerView = new girder.views.RegisterView({
        el: this.$('#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 () {
    if (girder.currentUser) {
        $("#login").addClass("hidden");
        $("#register").addClass("hidden");
        $("#name").removeClass("hidden");
        $("#logout").removeClass("hidden");
        $("#name").text("Logged in as " + girder.currentUser.get('firstName') + " " +
                        girder.currentUser.get('lastName'));

        // Do anything else you would like to do on login.
    } else {
        $("#login").removeClass("hidden");
        $("#register").removeClass("hidden");
        $("#name").addClass("hidden");
        $("#logout").addClass("hidden");

        // Do anything else you would like to do on logout.
    }
});

// Check who is logged in initially.
girder.restRequest({
    path: 'user/authentication',
    error: null
}).done(function () {
    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**.

3.5 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
```

CHAPTER 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.

4.1 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.

4.1.1 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.

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.

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.

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 `GirderClient.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 GirderClient 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.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*