giftless

Release 0.5.0

Shahar Evron

Apr 26, 2022

CONTENTS:

Giftless a Python implementation of a Git LFS Server. It is designed with flexibility in mind, to allow pluggable storage backends, transfer methods and authentication methods.

CHAPTER

ONE

INSTALLATION / DEPLOYMENT

You can install and run Giftless in different ways, depending on your needs:

1.1 Running from Docker image

Giftless is available as a Docker image available from Docker Hub

To run the latest version of Giftless in HTTP mode, listening on port 8080, run:

```
$ docker run --rm -p 8080:8080 datopian/giftless \
    -M -T --threads 2 -p 2 --manage-script-name --callable app \
    --http 0.0.0.0:8080
```

This will pull the image and run it.

Alternatively, to run in `WSGI` mode you can run:

```
$ docker run --rm -p 5000:5000 datopian/giftless
```

This will require an HTTP server such as *nginx* to proxy HTTP requests to it.

If you need to, you can also build the Docker image locally as described below.

1.2 Running from Pypi package

You can install Giftless into your Python environment of choice (3.7+) using pip. It is recommended to install Giftless into a virtual environment:

```
(venv) $ pip install uwsgi
(venv) $ pip install giftless
```

IMPORTANT: as of the time of writing, a bug in one of Giftless' dependencies requires that you explicitly install dependencies after installing using `pip`:

```
(venv) $ pip install -Ur https://raw.githubusercontent.com/datopian/giftless/master/
↪requirements.txt
```

Once installed, you can run Giftless locally with uWSGI:

```
# Run uWSGI (see uWSGI's manual for help on all arguments)
(venv) $ uwsgi -M -T --threads 2 -p 2 --manage-script-name \
    --module giftless.wsgi_entrypoint --callable app --http 127.0.0.1:8080
```

This will listen on port `8080`.

You should be able to replace `uwsgi` with any other WSGI server, such as `gunicorn`.

1.3 Running from source installation

You can install and run `giftless` from source:

```
$ git clone https://github.com/datopian/giftless.git

# Initialize a virtual environment
$ cd giftless
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
```

You can then proceed to run Giftless with a WSGI server as described above.

Note that for non-production use you may avoid using a WSGI server and rely on Flask's built in development server. This should **never** be done in a production environment:

```
(venv) $ ./flask-develop.sh
```

In development mode, Giftless will be listening on http://127.0.0.1:5000/

CHAPTER

TWO

HOW-TO GUIDES

This section includes several how-to guides designed to get you started with Giftless quickly.

2.1 Getting Started

This guide will introduce you to the basics of Giftless by getting it up and running locally, and seeing how it can interact with a local git repository.

2.1.1 Prerequisites

This tutorial assumes you have Python 3.7 or newer available as `python`. On some systems, you might need to replace `python` with `python3`.

2.1.2 Installing and Running Locally

Create a new directory for our tutorial, and set up a fresh virtual environment:

```
mkdir giftless && cd giftless
python -m venv .venv
source .venv/bin/activate
```

Then, proceed to *install giftless from pypi* as described in the installation guide.

Note: For this tutorial, we will be using Flask's built-in development server. This is *not suitable* for production use.

Once done, verify that Giftless can run:

```
# Run Giftless using the built-in development server
export FLASK_APP=giftless.wsgi_entrypoint
flask run
```

You should see something like:

```
Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

This means Giftless is up and running with some default configuration on *localhost* port *5000*, with the default configuration options.

Hit Ctrl+C to stop Giftless.

2.1.3 Basic Configuration

To configure Giftless, create a file named `giftless.conf.yaml` in the current directory with the following content:

```yaml
# Giftless configuration
AUTH_PROVIDERS:
  - giftless.auth.allow_anon:read_write
```

This will override the default read-only access mode, and allow open and full access to anyone, to any object stored with Giftless. Clearly this is not useful in a production setting, but for a local test this will do fine.

Run Giftless again, pointing to this new configuration file:

```
export GIFTLESS_CONFIG_FILE=giftless.conf.yaml
flask run
```

2.1.4 Interacting with git

We will now proceed to show how Giftless can interact with a local `git` repository, as a demonstration of how Git LFS works.

Keep Giftless running and open a new terminal window or tab.

Install the `lfs` Git extension

While having a local installation of `git-lfs` is not required to run Giftless, you will need it to follow this guide.

Run:

```
git lfs version
```

If you see an error indicating that `'lfs' is not a git command`, follow the Git LFS installation instructions here. On Linux, you may be able to simply install the `git-lfs` package provided by your distro.

Important: If you have git-lfs older than version 2.10, you will need to upgrade it to follow this tutorial, otherwise you may encounter some unexpected errors. Follow the instructions linked above to upgrade to the latest version.

Create a local "remote" repository

For the purpose of this tutorial, we will create a fake "remote" git repository on your local disk. This is analogous to a real-world remote repository such as GitHub or any other Git remote, but is simpler to set up.

```
mkdir fake-remote-repo && cd fake-remote-repo
git init --bare
cd ..
```

Of course, you may choose to use any other remote repository instead - just remember to replace the repository URL in the upcoming `git clone` command.

Create a local repository and push some file

Clone the remote repository we have just created to a local repository:

```
git clone fake-remote-repo local-repo
cd local-repo
```

Create some files and add them to git:

```
# This README file will be committed to Git as usual
echo "# This is a Giftless test" > README.md
# Let's also create a 1mb binary file which we'll want to store in Git LFS
dd if=/dev/zero of=1mb-blob.bin bs=1024 count=1024
git add README.md 1mb-blob.bin
```

Enable Git LFS and tell it to track `.bin` files:

```
git lfs install
git lfs track "*.bin"
```

This will actually create a file named `.gitattributes` in the root of your repository, with the following content:

```
*.bin filter=lfs diff=lfs merge=lfs -text
```

Tell Git LFS where to find the Giftless server. We will do that by using the `git config` command to write to the `.lfsconfig` file:

```
git config -f .lfsconfig lfs.url http://127.0.0.1:5000/my-organization/test-repo
```

`my-organization/test-repo` is an organization / repository prefix under which your files will be stored. Giftless requires all files to be stored under such prefix.

Tell git to track the configuration files we have just created. This will allow other users to have the same Git LFS configuration as us when cloning the repository:

```
git add .gitattributes .lfsconfig
```

Commit all the files we have staged:

```
git commit -m "Adding some files to track"
```

Finally, let's push our tracked files to Git LFS:

```
git push -u origin master
```

See your objects stored by Giftless locally

Switch over to the shell in which Giftless is running, and you will see log messages indicating that a file has just been pushed to storage and verified. This should be similar to:

```
INFO 127.0.0.1 - - "POST /my-organization/test-repo/objects/batch HTTP/1.1" 200 -
INFO 127.0.0.1 - - "PUT /my-organization/test-repo/objects/storage/
↪30e14955ebf1352266dc2ff8067e68104607e750abb9d3b36582b8af909fcb58 HTTP/1.1" 200 -
INFO 127.0.0.1 - - "POST /my-organization/test-repo/objects/storage/verify HTTP/1.1" 200␣
↪-
```

To further verify that the file has been stored by Giftless, we can list the files in our local Giftless storage directory:

```
ls -lR ../lfs-storage/
```

You should see:

```
../lfs-storage/my-organization/test-repo:
total 1024
-rw-rw-r-- 1 shahar shahar 1048576 Feb 28 12:08␣
↪30e14955ebf1352266dc2ff8067e68104607e750abb9d3b36582b8af909fcb58
```

You will notice a 1mb file stored in `../lfs-storage/my-organization/test-repo` - this is identical to our `1mb-blob.bin` file, but it is stored with its SHA256 digest as its name.

2.1.5 Summary

You have now seen Giftless used as both a Git LFS server, and as a storage backend. This should give you a basic sense of how to run Giftless, and how Git LFS servers interact with Git.

In a real-world scenario, you would typically have Giftless serve as the Git LFS server but not as a storage backend - storage will be off-loaded to a Cloud Storage service which has been configured for this purpose.

2.2 Using Google Cloud Storage as Backend

This guide will walk you through configuring Giftless to use Google Cloud Storage (GCS) as a storage backend. Using a cloud-based storage service such as GCS is highly recommended for production workloads and large files.

Our goal will be to run a local instance of Giftless, and interact with it using local `git` just as we did in the *quickstart guide*, but our LFS tracked files will be uploaded to, and downloaded from, GCS directly - Giftless will not be handling any file transfers.

A list of all provided storage backends is available *here*.

2.2.1 Prerequisites

- To use GCS you will need a Google Cloud account, and a Google Cloud project. Follow the Google Cloud Storage quickstart guide to create these.
- To follow this guide you will need to have the `gcloud` SDK installed locally and configured to use your project. Follow the installation guide, and then authorize your gcloud installation to access your project.
- If you already had `gcloud` installed before this tutorial, make sure you have configured `gcloud` to use the correct account and project before following this guide.

Important: Using Google Cloud may incur some charges. It is recommended to remove any resources created during this tutorial.

2.2.2 Set up a GCS Bucket and Service Account

GCS stores files (or "objects") in containers named *buckets*. Giftless will need read/write access to such a bucket via a *service account* - a software-only account with specific permissions.

NOTE: If you are familiar with Google Cloud Storage and are only interested in configuring Giftless to use it, and have a bucket and service account key at ready, you can skip this part.

Create a GCP service account

Create a GCP service account in the scope of our project:

```
gcloud iam service-accounts create giftless-test \
  --display-name "Giftless Test Account"
```

Then, run:

```
gcloud iam service-accounts list
```

The last command should list out the project's service account. Look for an email address of the form:

```
giftless-test@<yourproject>.iam.gserviceaccount.com
```

This address is the identifier of the account we have just created - we will need it in the next steps.

Create a GCS bucket and grant access to it

Create a bucket named `giftless-storage`:

```
gsutil mb gs://giftless-storage
```

Then grant our service account access to the bucket:

```
gsutil iam ch \
  serviceAccount:giftless-test@<yourproject>.iam.gserviceaccount.com:objectCreator,
↪objectViewer \
  gs://giftless-storage
```

Replace `giftless-test@<yourproject>.iam.gserviceaccount.com` with the email address copied above. This will grant the account read and write access to any object in the bucket, but will not allow it to modify the bucket itself.

Download an account key

In order to authenticate as our service account, Giftless will need a GCP Account Key JSON file. This can be created by running:

```
gcloud iam service-accounts keys create giftless-gcp-key.json \
  --iam-account=giftless-test@<yourproject>.iam.gserviceaccount.com
```

(again, replace `giftless-test@<yourproject>.iam.gserviceaccount.com` with the correct address)

This will create a file in the current directory named `giftless-gcp-key.json` - this is a secret key and should not be shared or stored in a non-secure manner.

2.2.3 Configure Giftless to use GCS

To use Google Cloud Storage as a storage backend and have upload and download requests be sent directly to GCS without passing through Giftless, we need to configure Giftless to use the `basic_external` transfer adapter with`GoogleCloudStorage` as storage backend.

Assuming you have followed the *getting started* guide to set up Giftless, edit your configuration YAML file (previously named `giftless.conf.yaml`) and add the `TRANSFER_ADAPTERS` section:

```
# Giftless configuration
AUTH_PROVIDERS:
  - giftless.auth.allow_anon:read_write

TRANSFER_ADAPTERS:
  basic:
    factory: giftless.transfer.basic_external:factory
    options:
      storage_class: giftless.storage.google_cloud:GoogleCloudStorage
      storage_options:
        project_name: giftless-tests
        bucket_name: giftless-storage
        account_key_file: giftless-gcp-key.json
```

Then, set the path to the configuration file, and start the local development server:

```
export GIFTLESS_CONFIG_FILE=giftless.conf.yaml
flask run
```

2.2.4 Upload and download files using local `git`

Follow the *quick start guide section titled "Interacting with git"* to see that you can push LFS tracked files to your Git repository. However, you will notice a few differences:

- The `git push` command may be slightly slower this time, as our 1mb file is upload to Google Cloud via the Internet and not over the loopback network.
- The Giftless logs will show only two lines, and not three - something like:

  ```
  INFO 127.0.0.1 - - "POST /my-organization/test-repo/objects/batch HTTP/1.1" 200 -
  INFO 127.0.0.1 - - "POST /my-organization/test-repo/objects/storage/verify HTTP/1.1
  ↪" 200 -
  ```

  This is because the `PUT` request to do the actual upload was sent directly to Google Cloud by `git-lfs`, and not to your local Giftless instance.
- You will not see any files stored locally this time

Behind the scenes, what happens with this setup is that when the Git LFS client asks Giftless to upload an object, Giftless will respond by providing the client with a URL to upload the file(s) to. This URL will be a pre-signed GCP URL, allowing temporary, limited access to write the specific file to our GCP bucket. The Git LFS client will then proceed to upload the file using that URL, and then call Giftless again to verify that the file has been uploaded properly.

Check that your object is in GCS

You can check that the object has been uploaded to your GCS bucket by running:

```
gsutil ls gs://giftless-storage/my-organization/test-repo/
```

You should see something like:

```
gs://giftless-storage/my-organization/test-repo/
↪30e14955ebf1352266dc2ff8067e68104607e750abb9d3b36582b8af909fcb58
```

Download Objects from Git LFS

To see how downloads work with Git LFS and Giftless, let's create yet another local clone of our repository. This simulates another user pulling from the same repository on a different machine:

```
cd ..
git clone fake-remote-repo other-repo
cd other-repo
```

You should now see that the `1mb-blob.bin` file exists in the other local repository, and is 1mb in size. The Gitless logs should show one more line, detailing the request made by `git-lfs` to request access to the file in storage. The file itself has been pulled from GCS.

2.2.5 Summary

In this guide, we have seen how to configure Giftless to use GCP as a storage backend. We have seen that Giftless, and other Git LFS servers, do not need (and in fact typically should not) serve as a file storage service, but in fact serve as a "gateway" to our storage backend.

The Google Cloud Storage backend has some additional options. See the full list of options for the Google Cloud Storage backend here

2.3 Setting Up JWT Authorization

This guide shows how Giftless could be set up to accept JWT tokens as a mechanism for both authentication and authorization, and shows some examples of manually generating JWT tokens and sending them to Giftless.

JWT tokens are useful because they allow an external service to verify the user's identity and grant it specific access permissions to objects stored by Giftless. The external service generates a token which can be verified by Giftless, but Giftless does not need to have any awareness of the granting party's authentication mechanism or authorization logic.

It is recommended that you read the *Authentication and Authorization overview* section in the documentation to get yourself familiar with some basic concepts.

This tutorial assumes you have at least completed the *Getting Started* guide, and have Giftless configured to store objects either locally or *in Google Cloud Storage*. The code samples assume you are running in the same directory and virtual environment in which Giftless was installed in previous tutorials.

2.3.1 Generate an RSA key pair

JWT tokens can be signed and encrypted using different algorithms, but one common and often useful pattern is to use an RSA public / private key pair. This allows distributing the public key, used for token verification, to multiple (potentially untrusted) services while token generation is done using a secret key.

For this tutorial we will generate an RSA key pair:

```
# Generate an RSA private key in PEM encoded format
ssh-keygen -t rsa -b 4096 -m PEM -f jwt-rs256.key

# Extract the public key to a PEM file
openssl rsa -in jwt-rs256.key -pubout -outform PEM -out jwt-rs256.key.pub
```

Do not enter any passphrase when generating the private key.

This will create two files in the current directory: `jwt-rsa256.key` which is the private key, and `jwt-rsa256.key.pub` which is the public key.

2.3.2 Configure Giftless to use JWT

To configure Giftless to require JWT tokens in requests, modify the `AUTH_PROVIDERS` section in your Giftless config file (`giftless.conf.yaml`) t0:

```
AUTH_PROVIDERS:
  - factory: giftless.auth.jwt:factory
    options:
      algorithm: RS256
      public_key_file: jwt-rs256.key.pub
```

Then, set the path to the configuration file, and start the local development server:

```
export GIFTLESS_CONFIG_FILE=giftless.conf.yaml
flask run
```

2.3.3 Generating a JWT Token

We now need to generate a JWT token that both authenticates us with Giftless, and carries some authorization information in the form of granted *scopes*. These tell Giftless what access level (e.g. read or write) the user has to which namespaces or objects.

In a production setting, JWT tokens will be generated by a special-purpose authorization service. For the purpose of this tutorial, we will manually generate tokens using `pyjwt` - a command line tool that comes with the PyJWT Python library:

Note: you can also use the debugging tool in https://jwt.io to generate tokens

```
pip install pyjwt  # this should already be installed in your Giftless virtual␣
↪environment
SCOPES="obj:my-organization/*"
```

(continues on next page)

(continued from previous page)

```
JWT_TOKEN=$(pyjwt --alg RS256 --key "$(cat jwt-rs256.key)" encode exp=+3600 sub=mr-robot␣
↪scopes="$SCOPES")
echo $JWT_TOKEN
```

This generates a JWT token identifying the user as `mr-robot`. The token is valid for 1 hour, and will grant both read and write access to every object under the `my-organization` namespace.

2.3.4 Using JWT tokens with Git LFS clients

Authenticating from custom Python code

giftless-client is a Python implementation of a Git LFS client with some Giftless specific extras such as support for the `multipart-basic` transfer mode.

To use the JWT token we have just generated with `giftless-client`, let's install it into our tutorial virtual environment:

```
pip install giftless-client
```

And create the following Python script in `lfs-client.py`:

```python
import os
import sys
from giftless_client import LfsClient


def main(file):
    token = os.getenv('JWT_TOKEN')  # assuming we set the env var above
    organization = 'my-organization'
    repo = 'my-repo'

    client = LfsClient(
        lfs_server_url='http://127.0.0.1:5000', # Git LFS server URL
        auth_token=token                        # JWT token
    )

    result = client.upload(file, organization, repo)
    print(f"Upload complete: Object ID {result['oid']}, {result['size']} bytes")

if __name__ == '__main__':
    with open(sys.argv[1], 'rb') as f:
        main(f)
```

Assuming you have set the `JWT_TOKEN` environment variable as described above, run it using:

```
export JWT_TOKEN
dd if=/dev/urandom of=random-file.bin bs=1024 count=1024 # Generate a random 1mb file
python lfs-client.py random-file.bin  # Upload it to storage via Git LFS
```

The output should be something like:

```
Upload complete: Object ID␣
↪7e3dd874a5475c946e441e17181cfcd8fac7760bb373d38b93aee9f9a93dce2e, 1048576 bytes
```

(with a different object ID value)

Note: this will fail if an hour or more has passed since we generated our token. In this case, simply regenerate the token and run again

Authenticating from Web-based Clients

One advantage of using JWT tokens for authentication and authorization is that they can be securely handed to Web clients, which in turn can use them to upload and download files securely from LFS-managed storage, directly from a Web application.

While we will not be demonstrating using Giftless to upload and download files from a browser, you can do this using JWT tokens to authenticate. giftless-client-js is a JavaScript client library which implements the Git LFS protocol with some Giftless specific "extras". You can look at the documentation of this library to see some examples of how to do this.

Important: To communicate with Giftless from a browser, you will most likely need to enable CORS support in Giftless, or deploy Giftless on the same scheme / host / port which serves your Web application.

Authenticating command line `git lfs`

Let's see if we can now use this token to push an object using command line `git-lfs`. Assuming you have followed the *Getting Started* tutorial, you should have a local repository checked out under `./local-repo`. We'll add another large file to this repo:

```
cd local-repo
dd if=/dev/zero of=512kb-blob.bin bs=1024 count=512
git add 512kb-blob.bin
git commit -m "Added a 512 kb file"
git push
```

At this stage you will be asked for a username for your Giftless server.

Giftless allows passing a JWT token using the `Basic` HTTP authentication scheme. This is designed specifically to support clients, such as `git`, that do not always support sending custom authentication tokens, and piggyback on the `Basic` scheme which is widely supported.

To use this functionality with command line `git`, simply use `_jwt` as your username, and the JWT token as your password.

While we will not cover this as part of this tutorial, this functionality can be automated by configuring a git credentials helper for your Giftless base URL.

2.3.5 Summary

In this guide we have demonstrated one of Giftless' built-in authentication and authorization modes - JWT tokens. We have covered how to configure Giftless to accept them, and demonstrated generating and using them from different client environments.

Next, you can:

- Learn more about the *JWT scopes accepted by Giftless*
- Get a better understanding of *authentication and authorization in Giftless*

CHAPTER

THREE

RUNTIME CONFIGURATION

3.1 Passing Configuration Options

Giftless can be configured by pointing it to a YAML configuration file when starting, or through the use of environment variables.

Note: Changes to any configuration options will only take effect when Giftless is restarted.

3.1.1 As a YAML file

Giftless will read configuration from a YAML file pointed by the GIFTLESS_CONFIG_FILE environment variable.

```
# create a config file
cat <<EOF > giftless.conf.yaml
AUTH_PROVIDERS:
  - giftless.auth.allow_anon:read_write
EOF

# start giftless
export GIFTLESS_CONFIG_FILE=giftless.conf.yaml
uwsgi --module giftless.wsgi_entrypoint --callable app --http 127.0.0.1:8080
```

3.1.2 As a YAML / JSON string passed as environment variable

If you prefer not to use a configuration file, you can pass the same YAML content as the value of the GIFTLESS_CONFIG_STR environment variable:

```
export GIFTLESS_CONFIG_STR="
AUTH_PROVIDERS:
  - giftless.auth.allow_anon:read_write
"
# Proceed to start Giftless
```

Since YAML is a superset of JSON, you can also provide a more compact JSON string instead:

```
export GIFTLESS_CONFIG_STR='{"AUTH_PROVIDERS":["giftless.auth.allow_anon:read_write"]}'
# Proceed to start Giftless
```

Important: If you provide both a YAML file (as `GIFTLESS_CONFIG_FILE`) and a literal YAML string (as `GIFTLESS_CONFIG_STR`), the two will be merged, with values from the YAML string taking precedence over values from the YAML file.

3.1.3 By overriding specific options using environment variables

You can override some specific configuration options using environment variables, by exporting an environment variable that starts with `GIFTLESS_CONFIG_` and appends configuration object keys separated by underscores.

This capability is somewhat limited and only works if:

- The configuration option value is expected to be a string
- The configuration option value is not contained in an array
- None of the configuration object keys in the value's hierarchy contain characters that are not accepted in environment variables, such as `-`

For example, the option specified in the configuration file as:

```
TRANSFER_ADAPTERS:
  basic:
    options:
      storage_class: giftless.storage.azure:AzureBlobsStorage
```

Can be overridden by setting the following environment variable:

```
GIFTLESS_CONFIG_TRANSFER_ADAPTERS_BASIC_OPTIONS_STORAGE_CLASS=
↪"mymodule:CustomStorageBackend"
# Start giftless ...
```

3.1.4 Using a `.env` file

If Giftless is started from a working directory that has a `.env` file, it will be loaded when Giftless is started and used to set environment variables.

3.2 Configuration Options

The following configuration options are accepted by Giftless:

3.2.1 `TRANSFER_ADAPTERS`

A set of transfer mode name -> transfer adapter configuration pairs. Controls transfer adapters and the storage backends used by them.

See the *Transfer Adapters* section for a full list of built-in transfer adapters and their respective options.

You can configure multiple Git LFS transfer modes, each with its own transfer adapter and configuration. The only transfer mode that is configured by default, and that is required by the Git LFS standard, is `basic` mode.

Each transfer adapter configuration value is an object with two keys:

- `factory` (required) - a string referencing a Python callable, in the form `package.module.submodule:callable`. This callable should either be an adapter class, or a factory callable that returns an adapter instance.
- `options` (optional) - a key-value dictionary of options to pass to the callable above.

3.2.2 `AUTH_PROVIDERS`

An ordered list of authentication and authorization adapters to load. Each adapter can have different options.

Auth providers are evaluated in the order that they are configured when a request is received, until one of them provides Giftless with a user identity. This allows supporting more than one authentication scheme in the same Giftless instance.

See the *Auth Providers* section for a full list of supported auth providers and their respective options.

Each auth provider can be specified either as a string of the form `package.module.submodule:callable`, referencing a Python callable that returns the provider instance, or as an object with the following keys:

- `factory` - a string of the same form referencing a callable
- `options` - key-value pairs of arguments to pass to the callable

3.2.3 `MIDDLEWARE`

An ordered list of custom WSGI middleware configuration. See *Using WSGI Middleware* for details and examples.

3.2.4 `PRE_AUTHORIZED_ACTION_PROVIDER`

Configures an additional single, special auth provider, which implements the `PreAuthorizedActionAuthenticator` interface. This is used by Giftless when it needs to generate URLs referencing itself, and wants to pre-authorize clients using these URLs. By default, the JWT auth provider is used here.

There is typically no need to override the default behavior.

3.2.5 `DEBUG`

If set to `true`, enables more verbose debugging output in logs.

CHAPTER

FOUR

USING WSGI MIDDLEWARE

Another way Giftless allows customizing its behavior is using standard WSGI middleware. This includes both publicly available middleware libraries, or your own custom WSGI middleware code.

4.1 Enabling Custom WSGI Middleware

To enable a WSGI middleware, add it to the `MIDDLEWARE` config section like so:

```yaml
MIDDLEWARE:
  - class: wsgi_package.wsgi_module:WSGICallable
    args: []  # List of ordered arguments to pass to callable
    kwargs: {}  # key-value pairs of keyword arguments to pass to callable
```

Where:

- `class` is a `<full module name>:<class or factory>` reference to a WSGI module and class name, or a callable that returns a WSGI object
- `args` is a list of arguments to pass to the specified callable
- `kwargs` are key-value pair of keyword arguments to pass to the specified callable.

The middleware module must be installed in the same Python environment as Giftless for it to be loaded.

4.2 Useful Middleware Examples

Here are some examples of solving specific needs using WSGI middleware:

4.2.1 HOWTO: Fixing Generated URLs when Running Behind a Proxy

If you have Giftless running behind a reverse proxy, and available publicly at a custom hostname / port / path / scheme that is not known to Giftless, you might have an issue where generated URLs are not accessible.

This can be fixed by enabling the `ProxyFix` Werkzeug middleware, which is already installed along with Giftless:

```yaml
MIDDLEWARE:
  - class: werkzeug.middleware.proxy_fix:ProxyFix
    kwargs:
      x_host: 1
```

(continues on next page)

(continued from previous page)

```
      x_port: 1
      x_prefix: 1
```

In order for this to work, you must ensure your reverse proxy (e.g. nginx) sets the right `X-Forwarded-*` headers when passing requests.

For example, if you have deployed giftless in an endpoint that is available to clients at `https://example.com/lfs`, the following nginx configuration is expected, in addition to the Giftless configuration set in the `MIDDLEWARE` section:

```
    location /lfs/ {
        proxy_pass http://giftless.internal.host:5000/;
        proxy_set_header X-Forwarded-Prefix /lfs;
    }
```

This example assumes Giftless is available to the reverse proxy at `giftless.internal.host` port 5000. In addition, `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto` are automatically set by nginx by default.

4.2.2 HOWTO: CORS Support

If you need to access Giftless from a browser, you may need to ensure Giftless sends proper CORS headers, otherwise browsers may reject responses from Giftless.

There are a number of CORS WSGI middleware implementations available on PyPI, and you can use any of them to add CORS headers control support to Giftless.

For example, you can enable CORS support using wsgi-cors-middleware:

```
(.venv) $ pip install wsgi_cors_middleware
```

Note: when using the Giftless Docker image, there is no need to install this middleware as it is already installed)

And then add the following to your config file:

```
MIDDLEWARE:
  - class: wsgi_cors_middleware:CorsMiddleware
    kwargs:
      origin: https://www.example.com
      headers: ['Content-type', 'Accept', 'Authorization']
      methods: ['GET', 'POST', 'PUT']
```

CHAPTER

FIVE

ARCHITECTURE AND COMPONENTS

Giftless is highly modular, and allows customization of most of its behavior through specialized classes, each responsible for a different aspect of the server's operation.

5.1 Transfer Adapters

Git LFS servers and clients can implement and negotiate different transfer adapters. Typically, Git LFS will only define a `basic` transfer mode and support that. `basic` is simple and efficient for direct-to-storage uploads for backends that support uploading using a single `PUT` request.

5.1.1 `basic` Transfer Mode

External Storage `basic` transfer adapter

The `basic_external` transfer adapter is designed to facilitate LFS `basic` mode transfers (the default transfer mode of Git LFS) for setups in which the storage backends supports communicating directly with the Git LFS client. That is, files will be uploaded or downloaded directly from a storage service that supports HTTP `PUT` / `GET` based access, without passing through Giftless. With this adapter, Giftless will not handle any file transfers - it will only be responsible for providing the client with access to storage.

This transfer adapter works with storage adapters implementing the `ExternalStorage` storage interface - typically these are Cloud storage service based backends.

Streaming `basic` transfer adapter

The `basic_streaming` transfer adapter facilitates LFS `basic` mode transfers in which Giftless also handles object upload, download and verification requests directly. This is less scalable and typically less performant than the `basic_external` adapter, as all data and potentially long-running HTTP requests must be passed through Giftless and its Python runtime. However, in some situations this may be preferable to direct-to-storage HTTP requests.

`basic_streaming` supports local storage, and also streaming requests from some Cloud storage service backends such as Azure and Google Cloud - although these tend to also support the `basic_external` transfer adapter.

5.1.2 Multipart Transfer Mode

To support more complex, and especially multi-part uploads (uploads done using more than one HTTP request, each with a different part of a large file) directly to backends that support that, Giftless adds support for a non-standard `multipart-basic` transfer mode.

NOTE: `basic-multipart` is a non-standard transfer mode, and will not be supported by most Git LFS clients; For a Python implementation of a Git LFS client library that does, see giftless-client.

Enabling Multipart Transfer Mode

You can enable multipart transfers by adding the following lines to your Giftless config file:

```yaml
TRANSFER_ADAPTERS:
  # Add the following lines:
  multipart-basic:
    factory: giftless.transfer.multipart:factory
    options:
      storage_class: giftless.storage.azure:AzureBlobsStorage
      storage_options:
        connection_string: "somesecretconnectionstringhere"
        container_name: my-multipart-storage
```

You must specify a `storage_class` that supports multipart transfers (implements the `MultipartStorage` interface). Currently, these are:

- `giftless.storage.azure:AzureBlobsStorage` - Azure Blob Storage

The following additional options are available for `multipart-basic` transfer adapter:

- `action_lifetime` - The maximal lifetime in seconds for signed multipart actions; Because multipart uploads tend to be of very large files and can easily take hours to complete, we recommend setting this to a few hours; The default is 6 hours.
- `max_part_size` - Maximal length in bytes of a single part upload. The default is 10MB.

See the specific storage adapter for additional backend-specific configuration options to be added under `storage_options`.

5.2 Storage Backends

Storage Backend classes are responsible for managing and interacting with the system that handles actual file storage, be it a local file system or a remote, 3rd party cloud based storage.

Storage Adapters can implement one or more of several interfaces, which defines the capabilities provided by the backend, and which *transfer adapters* the backend can be used with.

5.2.1 Types of Storage Backends

Each storage backend adapter can implement one or more of the following interfaces:

- `StreamingStorage` - provides APIs for streaming object upload / download through the Giftless HTTP server. Works with the `basic_streaming` transfer adapter.
- `ExternalStorage` - provides APIs for referring clients to upload / download objects using an external HTTP server. Works with the `basic_external` transfer adapter. Typically, these backends interact with Cloud Storage providers.
- `MultipartStorage` - provides APIs supporting the special `multipart-basic` transfer mode. Typically, these backends interact with Cloud Storage providers.
- `VerifiableStorage` - provides API for verifying that an object was uploaded properly. Most concrete storage adapters implement this interface.

5.2.2 Configuring the Storage Backend

Storage backend configuration is provided as part of the configuration of each Transfer Adapter. For example:

```yaml
TRANSFER_ADAPTERS:
  basic:  # <- the name of the transfer mode, you can have more than one
    factory: giftless.transfer.basic_external:factory
    options:
      storage_class: giftless.storage.google_cloud:GoogleCloudStorage
      storage_options:

    # add an example here
```

5.2.3 Built-In Storage Backends

Microsoft Azure Blob Storage

`giftless.storage.azure:AzureBlobStorage`

Modify your `giftless.yaml` file according to the following config:

```
    $ cat giftless.yaml

    TRANSFER_ADAPTERS:
      basic:
        factory: giftless.transfer.basic_external:factory
        options:
          storage_class: ..storage.azure:AzureBlobsStorage
          storage_options:
            connection_string: GetYourAzureConnectionStringAndPutItHere==
            container_name: lfs-storage
            path_prefix: large-files
```

Google Cloud Storage

`giftless.storage.google_cloud:GoogleCloudStorage`

To use Google Cloud Storage as a backend, you'll first need:

- A Google Cloud Storage bucket to store objects in
- an account key JSON file (see here).

The key must be associated with either a user or a service account, and should have read / write permissions on objects in the bucket.

If you plan to access objects from a browser, your bucket needs to have CORS enabled.

You can deploy the account key JSON file and provide the path to it as the `account_key_file` storage option:

```yaml
TRANSFER_ADAPTERS:
  basic:
    factory: giftless.transfer.basic_streaming:factory
    options:
      storage_class: giftless.storage.google_cloud:GoogleCloudStorage
      storage_options:
        project_name: my-gcp-project
        bucket_name: git-lfs
        account_key_file: /path/to/credentials.json
```

Alternatively, you can base64-encode the contents of the JSON file and provide it inline as `account_key_base64`:

```yaml
TRANSFER_ADAPTERS:
  basic:
    factory: giftless.transfer.basic_streaming:factory
    options:
      storage_class: giftless.storage.google_cloud:GoogleCloudStorage
      storage_options:
        project_name: my-gcp-project
        bucket_name: git-lfs
        account_key_base64: S0m3B4se64RandomStuff.....ThatI5Redac7edHeReF0rRead4b1lity==
```

Amazon S3 Storage

`giftless.storage.amazon_s3:AmazonS3Storage`

Modify your `giftless.yaml` file according to the following config:

```
    $ cat giftless.yaml

    TRANSFER_ADAPTERS:
      basic:
        factory: giftless.transfer.basic_external:factory
        options:
          storage_class: giftless.storage.amazon_s3:AmazonS3Storage
          storage_options:
            bucket_name: bucket-name
            path_prefix: optional_prefix
```

boto3 authentication

`AwsS3Storage` supports 3 ways of authentication defined in more detail in docs:

1. Environment variables
2. Shared credential file (~/.aws/credentials)
3. AWS config file (~/.aws/config)
4. Instance metadata service on an Amazon EC2 instance that has an IAM role configured (usually used in production).

Running updated yaml config with uWSGI

After configuring your `giftless.yaml` file, export it:

```
$ export GIFTLESS_CONFIG_FILE=giftless.yaml
```

You will need uWSGI running. Install it with your preferred package manager. Here is an example of how to run it:

```
    # Run uWSGI in HTTP mode on port 8080
    $ uwsgi -M -T --threads 2 -p 2 --manage-script-name \
        --module giftless.wsgi_entrypoint --callable app --http 127.0.0.1:8080
```

Notes

- If you plan to access objects directly from a browser (e.g. using a JavaScript based Git LFS client library), your GCS bucket needs to be CORS enabled.

Local Filesystem Storage

`giftless.storage.local:LocalStorage`

TBD

5.3 Authentication and Authorization Providers

5.3.1 Overview

Authentication and authorization in Giftless are pluggable and can easily be customized. While Giftless typically bundles together code that handles both authentication and to some degree authorization, the two concepts should be understood separately first in order to understand how they are handled by Giftless.

- *Authentication* (sometimes abbreviated here and in the code as `authn`) relates to validating the identity of the entity (person or machine) sending a request to Giftless
- *Authorization* (sometimes abbreviated as `authz`) relates to deciding, once an identity has been established, whether the requesting party is permitted to perform the requested operation

Note: In this guide and elsewhere we may refer to *auth* as a way of referring to both authentication and authorization in general, or where distinction between the two concepts is not important.

5.3.2 Provided Auth Modules

Giftless provides the following authentication and authorization modules by default:

- `giftless.auth.jwt:JWTAuthenticator` - uses JWT tokens to both identify the user and grant permissions based on scopes embedded in the token payload.
- `giftless.auth.allow_anon:read_only` - grants read-only permissions on everything to every request; Typically, this is only useful in testing environments or in very limited deployments.
- `giftless.auth.allow_anon:read_write` - grants full permissions on everything to every request; Typically, this is only useful in testing environments or in very limited deployments.

5.3.3 Configuring Authenticators

Giftless allows you to specify one or more auth module via the `AUTH_PROVIDERS` configuration key. This accepts a *list* of one or more auth modules. When a request comes in, auth modules will be invoked by order, one by one, until an identity is established.

For example:

```
AUTH_PROVIDERS:
  - factory: giftless.auth.jwt:factory
    options:
      algorithm: HS256
      private_key: s3cret,don'ttellany0ne
  - giftless.auth.allow_anon:read_only
```

The config block above defines 2 auth providers: first, the `JWT` auth provider will be tried. If it manages to produce an identity (i.e. the request contains an acceptable JWT token), it will be used. If the request does not cotain a `JWT` token, Giftless will fall back to the next provider - in this case, the `allow_anon:read_only` provider which will allow read-only access to anyone.

This allows servers to be set up to accept different authorization paradigms.

You'll notice that each item in the `AUTH_PROVIDERS` list can be either an object with `factory` and `options` keys - in which case Giftless will load the auth module by calling the `factory` Python callable (in the example above, the `factory` function in the `giftless.auth.jwt` Python module); Or, in simpler cases, it can be just a string (as in the case of our 2nd provider), which will be treated as a `factory` value with no options.

Read below for the `options` possible for specific auth modules.

5.3.4 JWT Authenticator

This authenticator authenticates users by accepting a well-formed JWT token in the Authorization header as a Bearer type token, or as the value of the `?jwt=` query parameter. Tokens must be signed by the right key, and also match in terms of audience, issuer and key ID if configured, and of course have valid course expiry / not before times.

Piggybacking on `Basic` HTTP auth

The JWT authenticator will also accept JWT tokens as the password for the `_jwt` user in `Basic` HTTP `Authorization` header payload. This is designed to allow easier integration with clients that only support Basic HTTP authentication.

You can disable this functionality or change the expected username using the `basic_auth_user` configuration option.

Configuration Options

The following options are available for the `jwt` auth module:

- `algorithm` (`str`): JWT algorithm to use, e.g. `HS256` (default) or `RS256`. Must match the algorithm used by your token provider
- `public_key` (`str`): Public key string, used to verify tokens signed with any asymmetric algorithm (i.e. all algorithms except `HS*`); Optional, not needed if a symmetric algorithm is in use.
- `public_key_file` (`str`): Path to file containing the public key. Specify as an alternative to `public_key`.
- `private_key` (`str`): Private key string, used to verify tokens signed with a symmetric algorithm (i.e. `HS*`); Optional, not needed if an asymmetric algorithm is in use.
- `public_key_file` (`str`): Path to file containing the private key. Specify as an alternative to `private_key`.
- `leeway` (`int`): Key expiry time leeway in seconds (default is 60); This allows for a small clock time skew between the key provider and Giftless server
- `key_id` (`str`): Optional key ID string. If provided, only keys with this ID will be accepted.
- `basic_auth_user` (`str`): Optional HTTP Basic authentication username to look for when piggybacking on Basic authentication. Default is `_jwt`. Can be set to `None` to disable inspecting `Basic` auth headers.

Options only used when module used for generating JWT tokens

The following options are currently only in use when the module is used for generating tokens for self-signed requests (i.e. not as an `AUTH_PROVIDER`, but as a `PRE_AUTHORIZED_ACTION_PROVIDER`):

- `default_lifetime` (`int`): lifetime of token in seconds
- `issuer` (`str`): token issuer (optional)
- `audience` (`str`): token audience (optional)

JWT Authentication Flow

A typical flow for JWT is:

1. There is an external *trusted* system that can generate and sign JWT tokens and Giftless is configured to verify and accept tokens signed by this system
2. User is logged in to this external system
3. A JWT token is generated and signed by this system, granting permission to specific scopes applicable to Giftless
4. The user sends the JWT token along with any request to Giftless, using either the `Authorization: Bearer ...` header or the `?jwt=...` query parameter
5. Giftless validates and decodes the token, and proceeds to grant permissions based on the `scopes` claim embedded in the token.

To clarify, it is up to the 3rd party identity / authorization provider to decide, based on the known user identity, what scopes to grant.

Scopes

Beyond authentication, JWT tokens may also include authorization payload in the "scopes" claim.

Multiple scope strings can be provided, and are expected to have the following structure:

```
obj:{org}/{repo}/{oid}:{subscope}:{actions}
```

or:

```
obj:{org}/{repo}/{oid}:{actions}
```

Where:

- `{org}` is the organization of the target object
- `{repo}` is the repository of the target object. Omitting or replacing with `*` designates we are granting access to all repositories in the organization
- `{oid}` is the Object ID. Omitting or replacing with `*` designates we are granting access to all objects in the repository
- `{subscope}` can be `metadata` or omitted entirely. If `metadata` is specified, the scope does not grant access to actual files, but to metadata only - e.g. objects can be verified to exist but not downloaded.
- `{actions}` is a comma separated list of allowed actions. Actions can be `read`, `write` or `verify`. If omitted or replaced with a `*`, all actions are permitted.

Examples

Some examples of decoded tokens (note that added comments are not valid JSON):

```
{
  "exp": 1586253890,           // Token expiry time
  "sub": "a-users-id",         // Optional user ID
  "iat": 1586253590,           // Optional, issued at
  "nbf": 1586253590,           // Optional, not valid before
  "name": "User Name",         // Optional, user's name
```

(continues on next page)

(continued from previous page)

```
  "email": "user@example.com", // Optional, user's email
  "scopes": [
    // read a specific object
    "obj:datopian/somerepo/
↪6adada03e86b154be00e25f288fcadc27aef06c47f12f88e3e1985c502803d1b:read",

    // read the same object, but do not limit to a specific prefix
    "obj:6adada03e86b154be00e25f288fcadc27aef06c47f12f88e3e1985c502803d1b:read",

    // full access to all objects in a repo
    "obj:datopian/my-repo/*",

    // Read only access to all repositories for an organization
    "obj:datopian/*:read",

    // Metadata read only access to all objects in a repository
    "obj:datopian/my-repo:meta:verify",
  ]
}
```

Typically, a token will include a single scope - but multiple scopes are allowed.

Rejected and Ignored Tokens

This authenticator will pass on the attempt to authenticate if no token was provided, or it is not a JWT token, or if a key ID is configured and a provided JWT token does not have the matching "kid" head claim (this allows chaining multiple JWT authenticators if needed).

However, if a matching but invalid token was provided, a `401 Unauthorized` response will be returned. "Invalid" means a token with audience or issuer mismatch (if configured), an expiry time in the past, or a "not before" time in the future, or, of course, an invalid signature.

Additional Parameters

The `leeway` parameter allows for providing a leeway / grace time to be considered when checking expiry times, to cover for clock skew between servers.

5.3.5 Understanding Authentication and Authorization Providers

This part is more abstract, and will help you understand how Giftless handles authentication and authorization in general. If you want to create a custom auth module, or better understand how provided auth modules work, read on.

Giftless' authentication and authorization module defines two key interfaces for handling authentication and authorization:

Authenticators

Authenticator classes are subclasses of `giftless.auth.Authenticator`. One or more authenticators can be configured at runtime, and each authenticator can try to obtain a valid user identity from a given HTTP request.

Once an identity has been established, an `Identity` (see below) object will be returned, and it is the role of the Authenticator class to populate this object with information about the user, such as their name and email, and potentially, information on granted permissions.

Multiple authenticators can be chained, so that if one authenticator cannot find a valid identity in the request, the next authenticator will be called. If no authenticator manages to return a valid identity, by default a `401 Unauthorized` response will be returned for any action, but this behavior can be modified via the `@Authentication.no_identity_handler` decorator.

Identity

Very simply, an `Identity` object encapsulates information about the current user making the request, and is expected to have the following interface:

```
class Identity:
    name: Optional[str] = None
    id: Optional[str] = None
    email: Optional[str] = None

    def is_authorized(self, organization: str, repo: str, permission: Permission, oid:␣
↪Optional[str] = None) -> bool:
        """Tell if user is authorized to perform an operation on an object / repo
        """
        pass
```

Most notably, the `is_authorized` method will be used to tell whether the user, represented by the Identity object, is authorized to perform an action (one of the `Permission` values specified below) on a given entity.

Authorizer classes may use the default built-in `DefaultIdentity`, or implement an `Identity` subclass of their own.

Permissions

Giftless defines the following permissions on entites:

```
class Permission(Enum):
    READ = 'read'
    READ_META = 'read-meta'
    WRITE = 'write'
```

For example, if `Permission.WRITE` is granted on an object or a repository, the user will be allowed to write objects matching the granted organization / repository / object scope.

CHAPTER

SIX

DEVELOPMENT

This section is intended for developers aiming to modify, extend or contribute to Giftless, or that are interested in more extensive technical information.

6.1 Developer Guide

`giftless` is based on Flask, with the following additional libraries:

- Flask Classful for simplifying API endpoint implementation with Flask
- Marshmallow for input / output serialization and validation
- figcan for configuration handling

You must have Python 3.7 and newer set up to run or develop `giftless`.

6.1.1 Code Style

We use the following tools and standards to write `giftless` code:

- `flake8` to check your Python code for PEP8 compliance
- `import` statements are checked by `isort` and should be organized accordingly
- Type checking is done using `mypy`

Maximum line length is set to 120 characters.

6.1.2 Setting up a Virtual Environment

You should develop `giftless` in a virtual environment. We use `pip-tools` to manage both development and runtime dependencies.

The following snippet is an example of how to set up your virtual environment for development:

```
$ python3 -m venv .venv
$ . .venv/bin/activate

(.venv) $ pip install -r dev-requirements.txt
(.venv) $ pip-sync dev-requirements.txt requirements.txt
```

6.1.3 Running the tests

Once in a virtual environment, you can simply run `make test` to run all tests and code style checks:

```
$ make test
```

We use `pytest` for Python unit testing.

In addition, simple functions can specify some `doctest` style tests in the function docstring. These tests will be tested automatically when unit tests are executed.

6.1.4 Building a Docker image

Simply run `make docker` to build a `uWSGI` wrapped Docker image for Giftless. The image will be named `datopian/giftless:latest` by default. You can change it, for example:

```
$ make docker DOCKER_REPO=mycompany DOCKER_IMAGE_TAG=1.2.3
```

Will build a Docekr image tagged `mycompany/giftless:1.2.3`.

6.2 Specifications: Git LFS multipart-basic transfer mode

```
Version: 0.9.1
Date:    2020-10-09
Author:  Shahar Evron <shahar.evron@gmail.com>
```

This document describes the `multipart-basic` transfer mode for Git LFS. This is a protocol extension to Git LFS, defining a new transfer mode to be implemented by Git LFS clients and servers.

Giftless is to be the first implementation of `multipart-basic`, but we hope that this transfer mode can be implemented by other Git LFS implementations if it is found useful.

6.2.1 Reasoning

Many storage vendors and cloud vendors today offer an API to upload files in "parts" or "chunks", using multiple HTTP requests, allowing improved stability and performance. This is especially handy when files are multiple gigabytes in size, and a failure during the upload of a file would require re-uploading it, which could be extremely time consuming.

The purpose of the `multipart-basic` transfer mode is to allow Git LFS servers and client facilitate direct-to-storage uploads for backends supporting multipart or chunked uploads.

As the APIs offered by storage vendors differ greatly, `multipart-basic` transfer mode will offer abstraction over most of these complexities in hope of supporting as many storage vendors as possible.

6.2.2 Terminology

Throughout this document, the following terms are in use:

- *LFS Server* - The HTTP server to which the LFS `batch` request is sent
- *Client* or *LFS Client* - a client using the Git LFS protocol to push large files to storage via an LFS server
- *Storage Backend* - The HTTP server handling actual storage; This may or may not be the same server as the LFS server, and for the purpose of this document, typically it is not. A typical implementation of this protocol would have the Storage Backend be a cloud storage service such as Amazon S3 or Google Cloud Storage.

6.2.3 Design Goals

Must:

- Abstract vendor specific API and flow into a generic protocol
- Remain as close as possible to the `basic` transfer API
- Work at least with the multi-part APIs of Amazon S3, Google Cloud Storage and Azure Blob Storage,

Nice / Should:

- Define how uploads can be resumed by re-doing parts and not-redoing parts that were uploaded successfully (this may be vendor specific and not always supported)
- Offer a local storage adapter for testing purposes

6.2.4 High Level Protocol Specs

- The name of the transfer is `multipart-basic`
- Batch requests are the same as `basic` requests except that `{"transfers": ["multipart-basic", "basic"]}` is the expected transfers value. Clients **must** retain `basic` as the fallback transfer mode to ensure compatiblity with servers not implementing this extension.
- `{"operation": "download"}` replies work exactly like `basic` download request with no change
- `{"operation": "upload"}` replies will break the upload into several `actions`:
  - `init` (optional), a request to initialize the upload
  - `parts` (optional), zero or more part upload requests
  - `commit` (optional), a request to finalize the upload
  - `abort` (optional), a request to abort the upload and clean up all unfinished chunks and state
  - `verify` (optional), a request to verify the file is in storage, similar to `basic` upload verify actions
- Just like `basic` transfers, if the file fully exists and is committed to storage, no `actions` will be provided in the reply and the upload can simply be skipped
- Authentication and authorization behave just like with the `basic` protocol.

Request Objects

The `init`, `commit`, `abort` and each one of the `parts` actions contain a “request spec”. These are similar to `basic` transfer adapter `actions` but in addition to `href`, `header` and `expires_in` may also include `method` (optional) and `body` (optional) attributes, to indicate the HTTP request method and body. This allows the protocol to be vendor agnostic, especially as the format of `init` and `commit` requests tends to vary greatly between storage backends.

The default values for these fields depends on the action:

- `init` defaults to no body and `POST` method
- `commit` defaults to no body and `POST` method
- `abort` defaults to no body and `POST` method
- `parts` requests default to `PUT` method and should include the file part as body, just like with `basic` transfer adapters.

In addition, each `parts` request will include the `pos` attribute to indicate the position in bytes within the file in which the part should begin, and `size` attribute to indicate the part size in bytes. If `pos` is omitted, default to `0` (beginning of the file). If `size` is omitted, default to read until the end of file.

Request / Response Examples

Upload Batch Request

The following is a ~10mb file upload request:

```
{
  "transfers": ["multipart-basic", "basic"],
  "operation": "upload",
  "objects": [
    {
      "oid": "20492a4d0d84f8beb1767f6616229f85d44c2827b64bdbfb260ee12fa1109e0e",
      "size": 10000000
    }
  ]
}
```

Upload Batch Response

The following is a response for the same request, given an imaginary storage backend:

```
{
  "transfer": "multipart-basic",
  "objects": [
    {
      "oid": "20492a4d0d84f8beb1767f6616229f85d44c2827b64bdbfb260ee12fa1109e0e",
      "size": 10000000,
      "actions": {
        "parts": [
          {
            "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=0",
            "header": {
```

(continues on next page)

(continued from previous page)

```
              "Authorization": "Bearer someauthorizationtokenwillbesethere"
            },
            "pos": 0,
            "size": 2500000,
            "expires_in": 86400
          },
          {
            "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=1",
            "header": {
              "Authorization": "Bearer someauthorizationtokenwillbesethere"
            },
            "pos": 2500000,
            "size": 2500000,
            "expires_in": 86400
          },
          {
            "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=2",
            "header": {
              "Authorization": "Bearer someauthorizationtokenwillbesethere"
            },
            "pos": 5000000,
            "size": 2500000,
            "expires_in": 86400
          },
          {
            "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=3",
            "header": {
              "Authorization": "Bearer someauthorizationtokenwillbesethere"
            },
            "pos": 7500000,
            "expires_in": 86400
          }
        ],
        "commit": {
          "href": "https://lfs.mycompany.com/myorg/myrepo/multipart/commit",
          "authenticated": true,
          "header": {
            "Authorization": "Basic 123abc123abc123abc123abc123=",
            "Content-type": "application/vnd.git-lfs+json"
          },
          "body": "{\"oid\": \"20492a4d0d84\", \"size\": 10000000, \"parts\": 4, \
↪"transferId\": \"foobarbazbaz\"}",
          "expires_in": 86400
        },
        "verify": {
          "href": "https://lfs.mycompany.com/myorg/myrepo/multipart/verify",
          "authenticated": true,
          "header": {
            "Authorization": "Basic 123abc123abc123abc123abc123="
          },
          "expires_in": 86400
        },
```

(continues on next page)

(continued from previous page)

```
        "abort": {
          "href": "https://foo.cloud.com/storage/upload/20492a4d0d84",
          "authenticated": true,
          "header": {
            "Authorization": "Basic 123abc123abc123abc123abc123="
          },
          "method": "DELETE",
          "expires_in": 86400
        }
      }
    }
  ]
}
```

As you can see, the `init` action is omitted as will be the case with many backend implementations (we assume initialization, if needed, will most likely be done by the LFS server at the time of the batch request).

Chunk sizing

It is up to the LFS server to decide the size of each file chunk.

Uploaded Part Digest

Some storage backends will support, or even require, uploading clients to send a digest of the uploaded part as part of the request. This is a useful capability even if not required, as it allows backends to validate each part separately as it is uploaded.

To support this, `parts` request objects may include a `want_digest` value, which may be any value specified by RFC-3230 or RFC-5843 (the design for this feature is highly inspired by these RFCs).

RFC-3230 defines `contentMD5` as a special value which tells the client to send the Content-MD5 header with an MD5 digest of the payload in base64 encoding.

Other possible values include a comma-separated list of q-factor flagged algorithms, one of MD5, SHA, SHA-256 and SHA-512. Of one or more of these are specified, the digest of the payload is to be specified by the client as part of the Digest header, using the format specified by RFC-3230 section 4.3.2.

Clients, when receiving a `parts` object with a `want_digest` value, must include in the request to upload the part a digest of the part, using the `Content-MD5` HTTP header (if `contentMD5` is specified as a value), or `Digest` HTTP header for any other algorithm / `want_digest` value.

Digest Control Examples

Examples of a batch response with `want_digest` in the reply

With “contentMD5”:

```
{
  "actions": {
    "parts": [
      {
```

(continues on next page)

(continued from previous page)

```
        "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=3",
        "header": {
          "Authorization": "Bearer someauthorizationtokenwillbesethere"
        },
        "pos": 7500001,
        "want_digest": "contentMD5"
      }
    ]
  }
}
```

With sha-256 as a preferred algorithm, and md5 as a less preferred option if sha-256 is not possible:

```
{
  "actions": {
    "parts": [
      {
        "href": "https://foo.cloud.com/storage/upload/20492a4d0d84?part=3",
        "header": {
          "Authorization": "Bearer someauthorizationtokenwillbesethere"
        },
        "pos": 7500001,
        "want_digest": "sha-256;q=1.0, md5;q=0.5"
      }
    ]
  }
}
```

Example of part upload request send to the storage server

Following on the `want_digest` value specified in the last example, the client should now send the following headers to the server when uploading the part:

```
HTTP/1.1 PUT /storage/upload/20492a4d0d84?part=3
Authorization: Bearer someauthorizationtokenwillbesethere
Digest: SHA-256=thvDyvhfIqlvFe+A9MYgxAfm1q5=,MD5=qweqweqweqweqweqweqwe=
```

Or if `contentMD5` was specified:

```
HTTP/1.1 PUT /storage/upload/20492a4d0d84?part=3
Authorization: Bearer someauthorizationtokenwillbesethere
Content-MD5: qweqweqweqweqweqweqwe=
```

Expected HTTP Responses

For each of the `init`, `commit`, `abort` and `parts` requests sent by the client, the following responses are to be expected:

- Any response with a `20x` status code is to be considered by clients as successful. This ambiguity is by design, to support variances between vendors (which may use `200` or `201` to indicate a successful upload, for example).
- Any other response is to be considered as an error, and it is up to the client to decide whether the request should be retried or not. Implementors are encouraged to follow standard HTTP error status code guidelines.
- An error such as `HTTP 409` on `commit` requests could indicates that not all the file parts have been uploaded successfully, thus it is not possible to commit the file. In such cases, clients are encouraged to issue a new `batch` request to see if any parts need re-uploading.
- An error such as `HTTP 409` on `verify` requests typically indicates that the file could not be verified. In this case, clients may issue an `abort` request (if an `abort` action has been specified by the server), and then retry the entire upload. Another approach here would be to retry the `batch` request to see if any parts are missing, however in this case clients should take special care to avoid infinite re-upload loops and fail the entire process after a small number of attempts.

`batch` replies for partially uploaded content

When content was already partially uploaded, the server is expected to return a normal reply but omit request and parts which do not need to be repeated. If the entire file has been uploaded, it is expected that no `actions` value will be returned, in which case clients should simply skip the upload.

However, if parts of the file were successfully uploaded while others weren't, it is expected that a normal reply would be returned, but with less `parts` to send.

6.2.5 Storage Backend Implementation Considerations

Hiding initialization / commit complexities from clients

While `part` requests are typically quite similar between vendors, the specifics of multipart upload initialization and commit procedures are very specific to vendors. For this reason, in many cases, it will be up to the LFS server to take care of initialization and commit code. This is fine, as long as actual uploaded data is sent directly to the storage backend.

For example, in the case of Amazon S3:

- All requests need to have an "upload ID" token which is obtained in an initial request
- When finalizing the upload, a special "commit" request need to be sent, listing all uploaded part IDs.

These are very hard to abstract in a way that would allow clients to send them directly to the server. In addition, as we do not want to maintain any state in the server, there is a need to make two requests when finalizing the upload: one to fetch a list of uploaded chunks, and another to send this list to the S3 finalization endpoint.

For this reason, in many cases storage backends will need to tell clients to send the `init` and `commit` requests to the LFS server itself, where storage backend handler code will take care of initialization and finalization. It is even possible for backends to run some initialization code (such as getting an upload ID from AWS S3) during the initial `batch` request.

Falling back to `basic` transfer for small files

Using multipart upload APIs has some complexity and speed overhead, and for this reason it is recommended that servers implement a "fallback" to `basic` transfers if the uploaded object is small enough to handle in a single part.

Clients *should* support such fallback natively, as it "rides" on existing transfer method negotiation capabilities.

The server must simply respond with `{"transfer": "basic", ...}`, even if `mutipart-basic` was request by the client and *is supported* by the server in order to achieve this.

Request Lifetime Considerations

As multipart uploads tend to require much more time than simple uploads, it is recommended to allow for longer `"expires_in"` values than one would consider for `basic` uploads. It is possible that the process of uploading a single object in multiple parts may take several hours from `init` to `commit`.

CHAPTER

SEVEN

GIFTLESS APIS

7.1 Transfer Adapters

7.2 Storage Backend Interfaces

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex