# media-nommer Documentation

***Release 1.0dev***

**DUO Interactive, LLC**

February 10, 2015

Contents

media-nommer is a distributed media encoding system that gets its horsepower from Amazon AWS. This leaves you with a cheaper, more reliable alternative to some commercial paid encoding systems, since you are only paying for your Amazon AWS usage. The software is powered by a combination of Python, Twisted, and Boto, and is licensed under the flexible BSD License.

CHAPTER 1

# This may be useful if...

media-nommer is of interest to you if you need a way to encode media in a cheap, fast, and scalable manner. By using Amazon AWS's excellent EC2 service, you can automatically scale your number of encoder instances up to meet your demands. Of course, the entire process may be configured, limited, and adapted to meet your needs. Some example usage cases where media-nommer might be a good fit:

- Encoding user-uploaded media for serving from your site
- Bulk encoding large collections of videos

CHAPTER 2

# Learning more

To learn more about media-nommer, see the *An Introduction to media-nommer*.

**Project Status:** Beta

**License:** media-nommer is licensed under the BSD License.

These links may also be useful to you.

- Source repository: https://github.com/duointeractive/media-nommer
- Issue tracker: https://github.com/duointeractive/media-nommer/issues
- Mailing list: https://groups.google.com/forum/?hl=en#!forum/media-nommer
- IRC Room: #medianommer on FreeNode

CHAPTER 3

# Documentation

## 3.1 An Introduction to media-nommer

media-nommer is a Python-based, distributed media encoding system. It aims to implement much of the same services offered by commercial encoding services. The primary advantage afforded by media-nommer is that one only pays for what they use on Amazon AWS, with no additional price mark-up. It may also be completely customized to meet your needs.

### 3.1.1 Some Background

There are any number of encoding services around the web, some of which are cheap and very easy to use. We (DUO Interactive) were searching for a suitable fit for two different projects that an extreme amount of encoding scalability. However, the more we looked into it, the more we struggled with picking a service that had a good balance of affordability and permanence. We also were looking for a little more customizability in the encoding process.

Many encoding services use software such as FFmpeg, and encode media to codecs that they have not paid licensing fees for. If the service has any degree of success, it is only a matter of time before MPEG-LA and others come knocking to get their licensing fees. This often spells end-of-game for the smaller (often cheaper) services. On the other end of the spectrum, some 100% legal services have to raise their prices up to cover the large licensing fees, putting it outside the realm of what we'd like to pay. A good read on this subject can be found on the FFmpeg legal page.

For those of us who don't need to encode to royalty-encumbered formats, a cheaper, more customizable, and more permanent solution was to just write our own. The result of all of this was the creation of media-nommer, a Python-based media encoding system with scalability and flexibility in mind.

### 3.1.2 A high level overview

media-nommer is comprised of two primary componenets, *feederd* and *ec2nommerd*.

#### feederd

Distributed encoding requires a director or orchestrator to keep everything running as it should. This is where the *feederd* daemon comes into play. Here are some of the things it does:

- Manages your EC2 instances (which have the *ec2nommerd* daemon running on them), and intelligently spins up more instances according to your configuration and work load.
- Keeps track of the status of your encoding jobs, present and past.

- Exposes a simple JSON-based API that can be used to schedule jobs, and check on the status of those that are currently running or completed.

*feederd* may be ran on your current infrastructure, or on EC2. Your EC2 encoding nodes are never in direct contact with *feederd*, and instead communicate through Amazon AWS. This means no firewall holes need to be opened.

#### ec2nommerd

Where *feederd* is the manager, *ec2nommerd* is the worker. Your basic unit of work is an EC2 instance that runs *ec2nommerd*. This daemon simple checks a SQS queue for jobs that need to be encoded, and does so until the queue is empty. After a period of inactivity, your EC2 instances will terminate themselves, saving you money.

## 3.2 Installation

Due to the early state of this project, there is a good chance that parts of these instructions are out of date at any given time. If you run into any such issue please let us know on our issue tracker.

---

**Note:** The following instructions are directed at what your eventual production environment will need. See *Hacking on media-nommer* for details on setting up a development environment.

---

### 3.2.1 Assumptions

For the sake of sanity, good deployment practices, and uniformity, we're going to make some assumptions. It is perfectly fine if you'd like to deviate, but it will be *much* more difficult for us to help you:

- You are deploying or developing media-nommer in a virtualenv. We highly recommend that and virtualenvwrapper.
- You are running a flavor of Linux, BSD, Mac OS, or something POSIX compatible. While Windows support is achievable, it's not something we currently have the resources to maintain (any takers?).
- You have Python 2.y or later. Python 2.7 is preferred. Python 3.x is not currently supported.
- You have an Amazon AWS account, and can create S3 buckets.

### 3.2.2 Requirements

- Some flavor of Linux, Unix, BSD, Mac OS, or POSIX compliant OS.
- Python 2.6 or higher, with Python 2.7 recommended. Python 3.x is not supported (yet).

### 3.2.3 Installing

For the sake of clarity, the media-nommer Python package contains the sources for *feederd* and *ec2nommerd*. You will want to install this package on whatever machine you'd like to run *feederd* on. You do not need to do any setup work for *ec2nommerd*, as those are ran on an EC2 instance that is based off of an AMI that we have created for you.

Since we are not yet distributing media-nommer on PyPi, the easiest way is to install the package is through **pip**:

```
pip install txrestapi twisted
pip install --upgrade git+http://github.com/duointeractive/media-nommer.git#egg=media_nommer
```

**Note:** If you don't have access to **pip**, you may download a tarball/zip, from our GitHub project and install via the enclosed `setup.py`. See the `requirements.txt` within the project for dependencies.

We have to install Twisted before media-nommer because of an odd behavior with **pip**.

**Tip:** Any time that you upgrade or re-install Twisted, you must also re-install media-nommer.

### 3.2.4 Signing up for AWS

After you have installed the media-nommer Python package on your machine(s), you'll need to visit Amazon AWS and sign up for the following services:

- SimpleDB
- SQS
- EC2

It is important to understand that even if you already have an Amazon account, you need to sign up to each of these services specifically for your account to have access to said services. This is a very quick process, and typically involves looking over an agreement and accepting it.

**Tip:** Signing up for these services is outside the scope of this document. Please contact AWS support with questions regarding this step. Their community forums are also a great resource.

Fees are based on what you actually use, so signing up for these services will incur no costs unless you use them.

### 3.2.5 AWS Management Console stuff

After you've signed up for all of the necessary services, there are a few steps to work through within the AWS Management Console. Sign in and take care of the following.

#### Create a Security Group

You will need to create a `media_nommer` EC2 *Security Group* through the AWS management console for your EC2 instances to be part of. You don't need to add any rules, though, unless you want SSH access to the encoding nodes.

**Warning:** Failure to create an EC2 security group will result in media-nommer not being able to spawn EC2 instances, which means no encoding for you.

#### Create or import an SSH key pair

Click on the **EC2** tab and find the **Key Pairs** link on the navigation bar. You can then either create or import a key pair. Make sure to keep track of the name of your keypair, as you'll need to specify it in your media-nommer configuration later on.

**Warning:** Failure to create or import an SSH key pair will also lead to media-nommer being unable to spawn EC2 instances.

### 3.2.6 Configuring

Once media-nommer is installed, create a directory to store a few files in. Within said directory, create a file called `nomconf.py`. You will want to add these settings values (at minimum):

```
# The AWS SSH key pair to use for creating instances. This is just the
# name of it, as per your Account Security Credentials panel on AWS.
EC2_KEY_NAME = 'xxxxxxxxx'
# The AWS credentials used to access SimpleDB, SQS, and EC2.
AWS_ACCESS_KEY_ID = 'YYYYYYYYYYYYYYYYYYYY'
AWS_SECRET_ACCESS_KEY = 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
# The S3 bucket to store a copy of ``nomconf.py`` for nommer instances.
# NOTE: CREATE A BUCKET AND CHANGE THIS VALUE BEFORE YOU START!
CONFIG_S3_BUCKET = 'change-this-value'
# The AWS security groups to create EC2 instances under. They don't need
# any rules set.
EC2_SECURITY_GROUPS = ['media_nommer']
```

See `media_nommer.conf.settings` for a full list of settings (and their defaults) that you may override in your `nomconf.py`.

### 3.2.7 Starting feederd

You will now want to start *feederd* using whatever init script or init daemon you use. We have had good results with Supervisor, but you can use whatever you're comfortable with. Make sure that the server running *feederd* is reachable by the machines that will be sending API requests to start encoding jobs. Here is an example command string, assuming you're in your top-level `media-nommer` directory (not `media_nommer`):

```
PYTHONPATH=media_nommer twistd -n --pidfile=feederd.pid feederd
```

### 3.2.8 Using

From here, you'll want to select and start using a *Client API Libraries*. These are what communicate with *feederd*'s RESTful web API, and ultimately help get stuff done.

## 3.3 feederd

Distributed encoding is a complex task, with lots of moving parts and components. To create, monitor, and "motivate" our EC2 encoding instances during the process, we have `feederd`.

### 3.3.1 What is feederd

`feederd` is a Twisted plugin that runs as a daemon on the machine of your choice. This can be on your own bare metal server, or on an EC2 instance. The daemon keeps track of encoding jobs through Amazon SimpleDB, tells your EC2 encoder instances to get to work via Amazon SQS, and exposes a lightweight JSON web API for your own custom applications to schedule and manage jobs through.

### 3.3.2 Relationship with ec2nommerd

If `feederd` is the brains of the operation, *ec2nommerd* is the brawn. `feederd` simply saves job state data to SimpleDB and creates SQS queue entries to let your EC2 instances know that there's work to be done. Each of your

EC2 instances run an *ec2nommerd* daemon, which pulls jobs from the queue (or terminates the instance if there isn't any work to be done).

### 3.3.3 Intelligent scaling

Along with providing an entry point to start, manage, and track the encoding process, `feederd` also handles scaling your encoding cloud up as needed. Based on your configuration, `feederd` will look at the current list of jobs that need to be encoded in comparison to the number of encoding instances you currently have running on EC2. It will then decide whether it should spawn additional instances to handle the load.

EC2 instances are spawned from a public AMI that we maintain as part of the project. If you wish to create your own custom AMI, you may easily specify your own or clone and modify ours.

Automated scaling is an optional feature, and can be configured and restricted in a number of different ways. For example, perhaps you don't want any more than one or two EC2 instances running at any given time.

### 3.3.4 JSON API

To actually get jobs scheduled and work done, you will need to communicate with `feederd` through its *JSON API*.

### 3.3.5 Architectural notes

One of the requirements for media-nommer was that at no time would `feederd` and one of its *ec2nommerd* instances communicate directly. As noted earlier, job state information is saved and retrieved from SimpleDB by both `feederd` and *ec2nommerd* instances. SQS is used to delegate work to the *ec2nommerd* instances. This does a few things for us:

- We don't have to open a hole in our firewall in your data center or in your EC2 encoder nodes.
- If `feederd` or the server that hosts it goes down, your *ec2nommerd* instances will continue encoding until the last job is popped off the queue. After that, they may be configured to terminate themselves after a period of inactivity (to save you money).

There is a lot of code that `feederd` and *ec2nommerd* shares, which can all be found in the `media_nommer.core` module. This is best thought of as a lower level API that these two components use to save and retrieve job state information.

Source for `feederd` can be found in the `media_nommer.feederd` module.

## 3.4 ec2nommerd

If *feederd* is the manager, `ec2nommerd` is the worker. You can also think of the relationship in that *feederd* feeds your nommers.

### 3.4.1 What is ec2nommerd

`ec2nommerd` is a Twisted plugin that runs as a daemon on EC2 instances in the cloud. These instances are automatically created by *feederd*. If an `ec2nommerd` finds itself with available encoding capacity, it will hit an Amazon SQS queue to see if there is any work available. If it finds anything, it pulls the job details from SimpleDB and hands the job off to the appropriate *nommer* for encoding.

If, after a period of time, the daemon receives no work to be done, it can be configured to terminate itself to save money.

### 3.4.2 Nommers

Nommers are classes that contain all of the logic specific to different kinds of encoding workflows. For example, the `FFmpegNommer` nommer wraps the FFmpeg command for encoding. Each nommer sub-classes the `BaseNommer` class, which provides some foundational methods. View the source for these two classes for examples on how they work. You may sub-class or create your own nommer.

The following Nommer classes are currently included with media-nommer. See each for details on how the `job_options` key should look like in your API calls.

- `media_nommer.ec2nommerd.nommers.ffmpeg.FFmpegNommer`

## 3.5 Using feederd's JSON API

One of *feederd*'s roles is hosting a web-based JSON API for your custom applications to communicate through. With this API, you may do such things as:

- Schedule encoding jobs
- Check the state of existing encoding jobs (To be implemented)

By using a simple web-based API, you are free to either use one of the existing client API libraries, or write your own.

### 3.5.1 Client API Libraries

The following API client libraries are for forming and sending JSON queries to media-nommer's web API. This is important for sending and checking on encoding jobs. Your applications will use the API to make media-nommer do something (other than stare at you blankly).

If you create a library of your own, let us know via the issue tracker and we'll add yours to the list.

- media-nommer-api (Python)

### 3.5.2 API Call Reference

All API calls are sent via **POST** with the body being JSON. Responses are also JSON-formatted.

---

**Note:** You should send these calls to the host/port that your *feederd* is running on. This defaults to 8001, but may specify when you call the **twistd** command to start the daemon.

---

#### /job/submit/

This call submits a job for encoding. Here is an example call with POST body:

```
{
    "source_path": "s3://AWS_ID:AWS_SECRET_KEY@BUCKET/KEYNAME.mp4",
    "dest_path": "s3://AWS_ID:AWS_SECRET_KEY@OTHER_BUCKET/KEYNAME.mp4",
    "notify_url": "http://myapp.somewhere.com/encoding/job_done/",
    "job_options": {
        "nommer": "media_nommer.ec2nommerd.nommers.ffmpeg.FFmpegNommer",
        "options": [
            {
                'outfile_options': [
                    ('vcodec', 'libx264'),
```

```
                    ('preset', 'medium'),
                    ('vprofile', 'baseline'),
                    ('b', '400k'),
                    ('vf', "yadif,scale='640:trunc(ow/a/2)*2'"),
                    ('pass', '1'),
                    ('f', 'mp4'),
                    ('an', None),
                ],
            },
            {
                'outfile_options': [
                    ('vcodec', 'libx264'),
                    ('preset', 'medium'),
                    ('vprofile', 'baseline'),
                    ('b', '400k'),
                    ('vf', "yadif,scale='640:trunc(ow/a/2)*2'"),
                    ('pass', '2'),
                    ('acodec', 'libfaac'),
                    ('ab', '128k'),
                    ('ar', '48000'),
                    ('async', '480'),
                    ('ac', '2'),
                    ('f', 'mp4'),
                ],
                'move_atom_to_front': True,
            },
        ],
    },
}
```

To further elaborate:

- `source_path` is a URI to the media file you'd like to encode. In this example, we use an S3 URI.
- `dest_path` is the full URI to where the output will end up.
- `job_options` is where you provide specifics as to what you'd like media-nommer to do. The `nommer` key is used to select which *nommer* to use, and the `options` key within `job_options` is used to configure said *nommer* (the values depend on the nommer). In the example above, we've selected the `FFmpegNommer` nommer, which wraps the excellent FFmpeg. See the documentation for the various *Nommers* to see what `job_options` is used for.
- `notify_url` is optional, and may be omitted entirely. If specified, this URL is hit with a GET request when the encoding job completes.

The response to your request is also JSON-formatted, and looks like this:

```
{
    "success": true,
    "job_id": "1f40fc92da241694750979ee6cf582f2d5d7d28e1833"
}
```

If an error is encountered, `success` will be `false`, additional `message` and `error_code` keys will be set:

```
{
    "false": true,
    "error_code": "BADINPUT",
    "message": "Bad input file. Unable to encode."
}
```

CHAPTER 4

# Development docs

The following topics will be useful to you if you would like to help improve media-nommer.

## 4.1 Hacking on media-nommer

For those that are particularly ambitious, or generous enough to want to contribute to the project, the barrier to entry is reasonably low. We will go over a few topics at a high level to get you started.

### 4.1.1 Obtaining the source

The best way to obtain the source is through our git repository on GitHub. If you are going to do anything aside from poke at the code, I would *highly* recommend visiting our GitHub project and forking the project via the **Fork** button.

For those not familiar with GitHub, this will create a copy of the repository under your username which you have commit access to. From your repository, you can make modifications and send *Pull Requests* to the upstream project asking for your changes to be merged in. Meanwhile, you can keep your fork up to date with changes from upstream. This is a whole lot faster, easier, and more fun than the traditional patch juggling method.

If you'd like to skip the whole *forking* ordeal and just get a copy of our upstream repository on your machine, you'll probably want to do something like:

```
git clone https://github.com/duointeractive/media-nommer.git
```

This will leave you with a *media-nommer* directory in your current directory.

---

**Tip:** For those that aren't familiar with git and/or GitHub, see the excellent GitHub help with lots of helpful how-tos.

---

### 4.1.2 Installing additional dependencies

There are a few additional dependencies to install when doing development work. Since everyone is being good little programmers and working under virtualenv, it's just a matter of switching to said virtual environment and doing this from within your `media-nommer` dir:

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

Now you're set to develop, run unit tests, and build our documentation.

### 4.1.3 Configuration

You will now want to review the *Configuring* section of *Installation* document. You may create your `nomconf.py` in your `media-nommer` directory for convenience. Return to this document once you have finished following the configuration instructions.

### 4.1.4 Running and writing unit tests

Running unit tests is trivial with nose. **cd** to your `media-nommer` directory and just run:

```
nose
```

The cool thing about nose is that it will find anything that looks like a unit test throughout the entire codebase, without us having to tell it where to look.

As far as writing unit tests, we'd like to shoot for full coverage, so please do write tests for your changes. If you are working on something new, find the *tests.py* module neighboring the one you're working on (or create one if it doesn't already exist) and write your unit tests using the standard Python unittest module. If you need any examples of how our unit tests look, run this to find all of our *tests.py* modules from your root *media-nommer* directory:

```
find . -name tests.py
```

Look through these for a good idea of what we're looking for.

> **Warning:** We are very unlikely to accept code contributions without unit tests. It is understood that writing unit tests is boring, tedious, and un-fun, but it is a necessary evil for complex software.

### 4.1.5 Running feederd locally

---

**Note:** This is suitable for local testing and development, your actual deployment would probably omit the `-n` flag to allow daemonization, unless you're using something like Supervisor.

---

If you are in your `media-nommer` directory, you may run *feederd* locally by doing this:

```
PYTHONPATH=media_nommer twistd -n --pidfile=feederd.pid feederd
```

### 4.1.6 Running ec2nommerd locally

*ec2nommerd* is designed to run on your EC2 instances, and is not at all meant to run on anything else. While it will do so just fine in most cases, a few features (such as self-termination) obviously won't work.

If you are in your `media-nommer` directory, you may run *ec2nommerd* locally by doing this:

```
PYTHONPATH=media_nommer twistd -n --pidfile=ec2nommerd.pid ec2nommerd -l
```

> **Warning:** Make **sure** to include the *-l* flag or your daemon will just deadlock while trying to query a web server that is internal to AWS.

### 4.1.7 Code style

We mostly adhere to PEP8, and expect contributors to do the same. A few quick hi-lights:

- 80 columns width max when possible.
- Indents are 4 spaces, and not tabs. **No tabs allowed**.
- Avoid wildcard * imports unless absolutely necessary.
- No camelCase method names, use underscores and lowercase letters.
- Classes use CapWords.
- Global variables are ALL_UPPER_CASE.
- Comments and docstrings for just about everything. Even if you think it's obvious. It probably won't be a few weeks/months/years later.

### 4.1.8 Contributing code

After you have made modifications in your GitHub fork, you need only send a *Pull Request* to us via the aptly named **Pull Request** on your fork's project page. See GitHub's guide to forking. It's quick and easy, we promise.

### 4.1.9 "I'm stumped, help!"

The best way to get help right now is to submit an issue on the issue tracker. This is useful for questions, suggestions, and bugs.

### 4.1.10 Contributions are BSD-licensed

Important to note is the fact that all contributions to the media-nommer project are, like the project itself, BSD-licensed. Please make sure you or your employer are OK with this before contributing.

## 4.2 media-nommer module reference

The following pages are reference material for media-nommer itself. This is probably only interesting to you if you are adding new Nommers. If you simply want to get to encoding, you'll want to select and start using one of the *Client API Libraries*.

### 4.2.1 media_nommer.conf

This module contains settings-related things that are used by *ec2nommerd* and *feederd*. You will most likely be interested in the `settings` module within this one, as that's where all of the settings and their values reside.

When the *ec2nommerd* and *feederd* Twisted plugins start, they use the `update_settings_from_module()` to override the defaults in the `settings` module with those specified by the user, typically via a user-provided module named `nomconf.py` (though that name can change with command line arguments).

If you need access to settings, simply import the global settings like this:

```
from media_nommer.conf import settings
```

media_nommer.conf.**update_settings_from_module**(*settings_module*)
Given another module with settings in uppercase variables on the module, override the defaults in `media_nommer.conf.settings` with the values from the given module.

**Parameters** **settings_module** (*module*) – A module with settings as upper-case attributes set. This is typically `nomconf.py`, although the user can elect to name them something else.

### settings

This module contains default global settings. When *ec2nommerd* and *feederd* daemons are started, `media_nommer.conf.update_settings_from_module()` takes the settings that the user explicitly set and overrides these defaults.

You may override any of these global defaults in your `nomconf.py` file. You will need to at the least provide the following settings:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `EC2_KEY_NAME`

`media_nommer.conf.settings.`**`AWS_ACCESS_KEY_ID`** **= None**
Default: `None` **(User must provide)**

These AWS credentials are used for job state management via SimpleDB, and queueing with SQS.

`media_nommer.conf.settings.`**`AWS_SECRET_ACCESS_KEY`** **= None**
Default: `None` **(User must provide)**

These AWS credentials are used for job state management via SimpleDB, and queueing with SQS.

`media_nommer.conf.settings.`**`CONFIG_S3_BUCKET`** **= 'nommer_config'**
Default: `'nommer_config'`

The S3 bucket to store a copy of `nomconf.py` for nommer instances.

`media_nommer.conf.settings.`**`EC2_AMI_ID`** **= 'ami-eb558182'**
Default: `The latest upstream AMI compatible with this git revision.`

The AMI ID for the media-nommer EC2 instance.

`media_nommer.conf.settings.`**`EC2_INSTANCE_TYPE`** **= 'm1.large'**
Default: `'m1.large'`

The type of instance to run on. Must be at least `m1.large`. `t1.micro` and `t1.small` instances are *NOT* supported by the default AMI.

`media_nommer.conf.settings.`**`EC2_KEY_NAME`** **= None**
Default: `None` **(User must provide)**

The AWS SSH key name with which to launch the EC2 instances.

`media_nommer.conf.settings.`**`EC2_SECURITY_GROUPS`** **= ['media_nommer']**
Default: `['media_nommer']`

The AWS security groups to create EC2 instances under.

`media_nommer.conf.settings.`**`FEEDERD_ABANDON_INACTIVE_JOBS_THRESH`** **= 32400**
Default: `3600 * 24`

If a job sticks in an un-finished state after this long (in seconds), it is considered abandoned, and *feederd* will kill discard it.

`media_nommer.conf.settings.`**`FEEDERD_ALLOW_EC2_LAUNCHES`** **= True**
Default: `True`

When `True`, allow the launching of new EC2 encoder instances.

media_nommer.conf.settings.**FEEDERD_AUTO_SCALE_INTERVAL = 60**
Default: `60`

How often *feederd* should see if it needs to spawn additional EC2 instances.

media_nommer.conf.settings.**FEEDERD_JOB_STATE_CHANGE_CHECK_INTERVAL = 60**
Default: `60`

How often *feederd* will check for job state changes.

media_nommer.conf.settings.**FEEDERD_PRUNE_JOBS_INTERVAL = 300**
Default: `60 * 5`

How often *feederd* will check for abandoned or expired jobs.

media_nommer.conf.settings.**JOB_OVERFLOW_THRESH = 2**
Default: `2`

If the number of unfinished jobs exceeds our capacity (`MAX_ENCODING_JOBS_PER_EC2_INSTANCE` * `<Number of Active EC2 instances>`) by this number of jobs, look at starting new instances if we have not already exceeded `MAX_NUM_EC2_INSTANCES`.

media_nommer.conf.settings.**MAX_ENCODING_JOBS_PER_EC2_INSTANCE = 2**
Default: `2`

The maximum number of jobs that should ever run on a single EC2 instance at the same time.

media_nommer.conf.settings.**MAX_NUM_EC2_INSTANCES = 3**
Default: `3`

The maximum number of EC2 instances to run at a time.

media_nommer.conf.settings.**NOMMERD_HEARTBEAT_INTERVAL = 60**
Default: `60`

Used by EC2 nodes to determine how long to wait between sending a status update via SimpleDB. The node will also check for inactivity greater than the configured value in `NOMMERD_MAX_INACTIVITY`, and terminate itself if inactivity has exceeded that value, and `NOMMERD_TERMINATE_WHEN_IDLE` is `True`.

media_nommer.conf.settings.**NOMMERD_MAX_INACTIVITY = 600**
Default: `60 * 50`

How many seconds of inactivity (not working on any jobs) before an instance will terminate itself.

media_nommer.conf.settings.**NOMMERD_NEW_JOB_CHECK_INTERVAL = 60**
Default: `60`

An interval (in seconds) to wait between calls to AWS to check for new jobs.

media_nommer.conf.settings.**NOMMERD_QTFASTSTART_BIN_PATH = '/home/nom/.virtualenvs/media_nommer/bin/qt**
The path to the qtfaststart bin used by ec2nommerd.

media_nommer.conf.settings.**NOMMERD_TERMINATE_WHEN_IDLE = True**
Default: `True`

When `True`, allow the termination of idle EC2 instances based on the `NOMMERD_MAX_INACTIVITY` setting. It is important to keep in mind that you pay for an entire hour when you start an EC2 instance, so setting this timeout to anything below 10 minutes is probably a waste of money. This timeout is in seconds.

media_nommer.conf.settings.**SIMPLEDB_EC2_NOMMER_STATE_DOMAIN = 'media_nommer_ec2nommer_state'**
Default: `'media_nommer_ec2nommer_state'`

The SimpleDB domain for storing heartbeat information from the EC2 encoder instances.

media_nommer.conf.settings.**SIMPLEDB_JOB_STATE_DOMAIN = 'media_nommer'**
Default: `'media_nommer'`

The SimpleDB domain name for storing encoding job state in. For example, the date the job was created, the current state of the job (`PENDING`, `ENCODING`, `FINISHED`, etc.), and the Nommer being used to do the encoding.

media_nommer.conf.settings.**SQS_JOB_STATE_CHANGE_QUEUE_NAME = 'media_nommer_jstate'**
Default: `'media_nommer_jstate'`

The SQS queue used to notify *feederd* of changes in job state. For example, when a job goes from `PENDING` to `DOWNLOADING` or `ENCODING`.

media_nommer.conf.settings.**SQS_NEW_JOB_QUEUE_NAME = 'media_nommer'**
Default: `'media_nommer'`

The SQS queue used to notify *ec2nommerd* of new jobs.

media_nommer.conf.settings.**STORAGE_BACKENDS = {'s3': 'media_nommer.core.storage_backends.s3.S3Backend', 'h**
Default: (All included storage backends)

Storage backends. The protocol is the key, the value is the backend class used to work with said protocol.

#### utils

Various configuration-related utility methods.

media_nommer.conf.utils.**download_settings**(*nomconf_uri*)
Given the URI to a S3 location with a valid nomconf.py, download it to the current user's home directory.

**Tip:** This is used on the media-nommer EC2 AMIs. This won't run on local development machines.

**Parameters** **nomconf_uri** (*str*) – The URI to your setup's `nomconf.py` file. Make Sure to specify AWS keys and IDs if using the S3 protocol.

media_nommer.conf.utils.**upload_settings**(*nomconf_module*)
Given a user-defined nomconf module (already imported), push said file to the S3 conf bucket, as defined by settings.CONFIG_S3_BUCKET. This is used by the nommers that require access to the config, like FFmpegNommer.

**Parameters** **nomconf_module** (*module*) – The user's `nomconf` module. This may be called something other than `nomconf`, but the uploaded filename will always be `nomconf.py`, so the EC2 nodes can find it in your settings.CONFIG_S3_BUCKET.

### 4.2.2 media_nommer.core

The core module is home to code that is important to both the `media_nommer.ec2nommerd` and `media_nommer.feederd` top-level modules. This is best thought of as a lower level API that all of the pieces of the encoding system use to communicate and interact with one another.

**Note:** There should be nothing that is specific to just *ec2nommerd* or just *feederd* here if at all possible.

### storage_backends

Storage backends are used to handle download and uploading content to a number of different protocols in an abstracted manner.

media-nommer uses URI strings to represent media locations, whether it be the file to download and encode, or where the output should be uploaded to. A reference to the correct backend for a URI can be found using the `get_backend_for_uri()` function.

`media_nommer.core.storage_backends.`**`get_backend_for_protocol`**(*protocol*)
Given a protocol string, return the storage backend that has been tasked with serving said protocol.

**Parameters** **protocol** (*str*) – A protocol string like 'http', 'ftp', or 's3'.

**Returns** A storage backend for the specified protocol.

`media_nommer.core.storage_backends.`**`get_backend_for_uri`**(*uri*)
Given a URI string , return a reference to the storage backend class capable of interacting with the protocol seen in the URI.

**Parameters** **uri** (*str*) – The URI to find the appropriate storage backend for.

**Return type** `StorageBackend`

**Returns** A reference to the backend class to use with this URI.

### s3

This module contains an S3Backend class for working with URIs that have an s3:// protocol specified.

**class** `media_nommer.core.storage_backends.s3.`**`S3Backend`**
Abstracts access to S3 via the common set of file storage backend methods.

**classmethod** **`download_file`**(*uri*, *fobj*)
Given a URI, download the file to the `fobj` file-like object.

**Parameters**

- **uri** (*str*) – The URI of a file to download.
- **fobj** (*file*) – A file-like object to download the file to.

**Return type** file

**Returns** A file handle to the downloaded file.

**classmethod** **`upload_file`**(*uri*, *fobj*)
Given a file-like object, upload it to the specified URI.

**Parameters**

- **uri** (*str*) – The URI to upload the file to.
- **fobj** (*file*) – The file-like object to populate the S3 key from.

**Return type** `boto.s3.key.Key`

**Returns** The newly set boto key.

### job_state_backend

The job state backend provides a simple API for *feederd* and *ec2nommerd* to store and retrieve job state information. This can be everything from the list of currently un-finished jobs, to the down-to-the-minute status of individual jobs.

There are two pieces to the job state backend:

- The `EncodingJob` class is used to interact with and manipulate individual jobs and their state.
- The `JobStateBackend` class is used to operate as the topmost manager. It can track all of the currently running jobs, get new jobs from the SQS queue, and process state change notifications from SQS.

**class** `media_nommer.core.job_state_backend.`**`EncodingJob`**(*source_path*, *dest_path*, *nommer*, *job_options*, *unique_id=None*, *job_state='PENDING'*, *job_state_details=None*, *notify_url=None*, *creation_dtime=None*, *last_modified_dtime=None*)

Represents a single encoding job. This class handles the serialization and de-serialization involved when saving and loading encoding jobs to and from SimpleDB.

---

**Tip:** You generally won't be instantiating these objects yourself. To retrieve an existing job, you may use `JobStateBackend.get_job_object_from_id()`.

---

**Parameters**

- **source_path** (*str*) – The URI to the source media to encode.
- **dest_path** (*str*) – The URI to upload the encoded media to.
- **job_options** (*dict*) – The job options to pass to the Nommer in key/value format. See each Nommer's documentation for details on accepted options.
- **unique_id** (*str*) – The unique hash ID for this job. If `save()` is called and this value is `None`, an ID will be generated for this job.
- **job_state** (*str*) – The state that the job is in.
- **job_state_details** (*str*) – Any details to go along with whatever job state this job is in. For example, if job_state is *ERROR*, this keyword might contain an error message.
- **notify_url** (*str*) – The URL to hit when this job has finished.
- **creation_dtime** (*datetime.datetime*) – The time when this job was created.
- **last_modified_dtime** (*datetime.datetime*) – The time when this job was last modified.

**`is_finished`**()

Returns True if this job is in a finished state.

**Return type** bool

**Returns** `True` if the job is in a finished state, or `False` if not.

**`save`**()

Serializes and saves the job to SimpleDB. In the case of a newly instantiated job, also handles queueing the job up into the new job queue.

**Return type** str

**Returns** The unique ID of the job.

**set_job_state**(*job_state*, *details=None*)
Sets the job's state and saves it to the backend. Sends a notification to *feederd* to re-load the job's data from SimpleDB via SQS.

**Parameters**

- **job_state** (*str*) – The state to set the job to.
- **job_state_details** (*str*) – Any details to go along with whatever job state this job is in. For example, if job_state is *ERROR*, this keyword might contain an error message.

**class** media_nommer.core.job_state_backend.**JobStateBackend**
Abstracts storing and retrieving job state information to and from SimpleDB. Jobs are represented through the EncodingJob class, which are instantiated and returned as needed.

**FINISHED_STATES = ['FINISHED', 'ERROR', 'ABANDONED']**
Any jobs in the following states are considered "finished" in that we won't do anything else with them. This is a list of strings.

**JOB_STATES = ['PENDING', 'DOWNLOADING', 'ENCODING', 'UPLOADING', 'FINISHED', 'ERROR', 'ABANDO**
All possible job states as a list of strings.

**classmethod get_job_object_from_id**(*unique_id*)
Given a job's unique ID, return an EncodingJob instance.

TODO: Make this raise a more specific exception.

**Parameters** **unique_id** (*str*) – An EncodingJob's unique ID.

**classmethod get_unfinished_jobs**()
Queries SimpleDB for a list of pending jobs that have not yet been finished.

**Return type** list

**Returns** A list of unfinished EncodingJob objects.

**classmethod pop_new_jobs_from_queue**(*num_to_pop*)
Pops any new jobs from the job queue.

> **Warning:** Once jobs are popped from a queue and delete() is ran on the message, they are gone for good. Be careful to handle errors in the methods higher on the call stack that use this method.

**Parameters** **num_to_pop** (*int*) – Pop up to this many jobs from the queue at once. This can be up to 10, as per SimpleDB limitations.

**Return type** list

**Returns** A list of EncodingJob objects.

**classmethod pop_state_changes_from_queue**(*num_to_pop*)
Pops any recent state changes from the queue.

> **Warning:** Once jobs are popped from a queue and delete() is ran on the message, they are gone for good. Be careful to handle errors in the methods higher on the call stack that use this method.

**Parameters** **num_to_pop** (*int*) – Pop up to this many jobs from the queue at once. This can be up to 10, as per SimpleDB limitations.

**Return type** list

**Returns** A list of EncodingJob objects.

**classmethod** **wipe_all_job_data**()
Deletes the SimpleDB domain and empties the SQS queue. These are both used to store and communicate job state data.

**Return type** bool

**Returns** `True` if successful. `False` if not.

### 4.2.3 media_nommer.ec2nommerd

The feederd module contains Twisted daemon that monitors queues, creates EC2 instances, schedules encoding jobs, handles response pings from external encoding services, and notifies your external applications when encoding has completed.

---

**Note:** Only more general management logic should live in this module. Anything specific like encoding commands or EC2 management should live elsewhere. feederd is just the orchestrator.

---

#### nommers

Nommers are classes that EC2 instances use with ec2nommerd to consume and produce media in the desired format. These usually involve wrapping external commands, setting job state details as the process continues, and uploading the results.

##### base_nommer

Classes in this module serve as a basis for Nommers. This should be thought of as a protocol or a foundation to assist in maintaining a consistent API between Nommers.

**class** media_nommer.ec2nommerd.nommers.base_nommer.**BaseNommer**(*job*)
This is a base class that can be sub-classed by each Nommer to serve as a foundation. Required methods raise a NotImplementedError exception by default, unless overridden by child classes.

**Variables** **job** (*EncodingJob*) – The encoding job this nommer is handling.

**download_source_file**()
Download the source file to a temporary file.

**onomnom**()
Start nomming. If you're going to override this, make sure to follow the same basic job state updating flow if possible.

**upload_to_destination**(*fobj*)
Upload the output file to the destination specified by the user.

**wrapped_set_job_state**(**args*, ***kwargs*)
Wraps set_job_state() to perform extra actions before and/or after job state updates.

**Parameters** **new_state** (*str*) – The job state to set.

##### ffmpeg

Contains a class used for nomming media on EC2 via FFmpeg. This is used in conjunction with the ec2nommerd Twisted plugin.

**class** media_nommer.ec2nommerd.nommers.ffmpeg.**FFmpegNommer**(*job*)

This *Nommer* is used to encode media with the excellent FFmpeg utility.

**Example API request**

Below is an example API request. The `job_options` dict that is passed to the *media_nommer.feederd* JSON API is the important part that is specific to this nommer.:

```
{
    'source_path': 'some_video.mp4',
    'dest_path': 'some_video_hqual.mp4',
    'notify_url': 'http://somewhere.com:8000/job_state_pingback',
    'job_options': {
        'nommer': 'media_nommer.ec2nommerd.nommers.ffmpeg.FFmpegNommer',
        # This options key has ffmpeg command line arguments for a 2-pass
        # encoding. If you're doing single pass, you'd only have one
        # dict in this list.
        'options': [
            # First pass command specification.
            {
                # Just documenting this here so its existence is known.
                'infile_options': [],
                # Fed to ffmpeg as command line flags. A None key means
                # that just the flag name is provided with no arg.
                'outfile_options': [
                    ('threads', 0),
                    ('vcodec', 'libx264'),
                    ('preset', 'medium'),
                    ('profile', 'baseline'),
                    ('b', '400k'),
                    ('vf', 'yadif,scale=640:-1'),
                    # This denotes the first pass in ffmpeg.
                    ('pass', '1'),
                    ('f', 'mp4'),
                    ('an', None),
                ],
            },
            # Second pass command specification.
            {
                'outfile_options': [
                    ('threads', 0),
                    ('vcodec', 'libx264'),
                    ('preset', 'medium'),
                    ('profile', 'baseline'),
                    ('b', '400k'),
                    ('vf', 'yadif,scale=640:-1'),
                    # Notice that this is now 2, for the second pass.
                    ('pass', '2'),
                    ('acodec', 'libfaac'),
                    ('ab', '128k'),
                    ('ar', '48000'),
                    ('ac', '2'),
                    ('f', 'mp4'),
                ],
            },
        ], # end options list, max of 2 passes.
    }, # end job_options
}
```

To show how this would be put together, here is the command that would be ran for the first pass:

```
ffmpeg -y -i some_video.mp4 -threads 0 -vcodec libx264 -preset medium -profile baseline -b 400k
```

Note that the `an` key in the `outfile_options` list of the first pass above has a `None` value. You'll need to do this for flags or options that don't require a value.

### interval_tasks

This module contains tasks that are executed at intervals, and is imported at the time the server is started. The intervals at which the tasks run are configurable via `media_nommer.conf.settings`.

All functions prefixed with `task_` are task functions that are registered with the Twisted reactor. All functions prefixed with `threaded_` are the interesting bits that actually do things.

`media_nommer.ec2nommerd.interval_tasks.`**`register_tasks`**`()`
Registers all tasks. Called by the *ec2nommerd* Twisted plugin.

`media_nommer.ec2nommerd.interval_tasks.`**`task_check_for_new_jobs`**`()`
Looks at the number of currently active threads and compares it against the `MAX_ENCODING_JOBS_PER_EC2_INSTANCE` setting. If we are under the max, fire up another thread for encoding additional job(s).

The interval at which *ec2nommerd* checks for new jobs is determined by the `NOMMERD_NEW_JOB_CHECK_INTERVAL` setting.

Calls `threaded_encode_job()` for any jobs to encode.

`media_nommer.ec2nommerd.interval_tasks.`**`task_heartbeat`**`()`
Checks in with feederd in a non-blocking manner via `threaded_heartbeat()`.

Calls `threaded_heartbeat()`.

`media_nommer.ec2nommerd.interval_tasks.`**`threaded_encode_job`**`(`*`job`*`)`
Given a job, run it through its encoding workflow in a non-blocking manner.

`media_nommer.ec2nommerd.interval_tasks.`**`threaded_heartbeat`**`()`
Fires off a threaded task to check in with feederd via SimpleDB. There is a domain that contains all of the running EC2 instances and their unique IDs, along with some state data.

The interval at which heartbeats occur is determined by the `NOMMERD_HEARTBEAT_INTERVAL <media_nommer.conf.settings.NOMMERD_HEARTBEAT_INTERVAL` setting.

### node_state

Contains the `NodeStateManager` class, which is an abstraction layer for storing and communicating the status of EC2 nodes.

**class** `media_nommer.ec2nommerd.node_state.`**`NodeStateManager`**
Tracks this node's state, reports it to *feederd*, and terminates itself if certain conditions of inactivity are met.

**classmethod** **`contemplate_termination`**`(`*`thread_count_mod=0`*`)`
Looks at how long it's been since this worker has done something, and decides whether to self-terminate.

**Parameters** **thread_count_mod** (*int*) – Add this to the amount returned by the call to `get_num_active_threads()`. This is useful when calling this method from a non-encoder thread.

**Return type** bool

**Returns** `True` if this instance terminated itself, `False` if not.

**classmethod get_instance_id**(*is_local=False*)
Determine this EC2 instance's unique instance ID. Lazy load this, and avoid further re-queries after the first one.

**Parameters** **is_local** (*bool*) – When True, don't try to hit EC2's meta data server, When False, just make up a unique ID.

**Return type** str

**Returns** The EC2 instance's ID.

**classmethod get_num_active_threads**()
Checks the reactor's threadpool to see how many threads are currently working. This can be used to determine how busy this node is.

**Return type** int

**Returns** The number of active threads.

**classmethod i_did_something**()
Pat ourselves on the back each time we do something.

Used for determining whether this node's continued existence is necessary anymore in `contemplate_termination()`.

**classmethod is_ec2_instance**()
Determine whether this is an EC2 instance or not.

**Return type** bool

**Returns** `True` if this is an EC2 instance, `False` if otherwise.

**last_dtime_i_did_something = datetime.datetime(2015, 2, 10, 19, 34, 10, 60820)**

**classmethod send_instance_state_update**(*state='ACTIVE'*)
Sends a status update to feederd through SimpleDB. Lets the daemon know how many jobs this instance is crunching right now. Also updates a timestamp field to let feederd know how long it has been since the instance's last check-in.

**Parameters** **state** (*str*) – If this EC2 instance is anything but `ACTIVE`, pass the state here. This is useful during node termination.

### 4.2.4 media_nommer.feederd

This module contains the logic for the Twisted daemon that oversees the encoding process, *feederd*. Only things that are specific to this daemon should reside within this module. Anything that could conceivably also be useful to `media_nommer.ec2nommerd` should probably be in `media_nommer.core`.

#### ec2_instance_manager

Contains the `EC2InstanceManager` class, which helps manage the currently active instances.

**class** media_nommer.feederd.ec2_instance_manager.**EC2InstanceManager**
This class managers and creates EC2 instances from the configured encoder AMI. Additional instances are spawned based on the size of the job queue in relation to the amount of manpower available at any point in time.

**classmethod get_instances**()
Returns a list of boto `Instance` objects matching the media-nommer AMI, as per `media_nommer.conf.settings.EC2_AMI_ID`. Also filters only running instances.

**Return type** list

**Returns** A list of `boto.ec2.instance.Instance` objects representing currently active media-nommer *ec2nommerd* instances.

**classmethod spawn_if_needed**()
Spawns additional EC2 instances if needed.

**Return type** `boto.ec2.instance.Reservation` or `None`

**Returns** If instances are spawned, return a boto Reservation object. If no instances are spawned, `None` is returned.

**classmethod spawn_instances**(*num_instances*)
Spawns the number of instances specified.

**Parameters** **num_instances** (*int*) – The number of instances to spawn.

**Return type** `boto.ec2.instance.Reservation`

**Returns** A boto Reservation for the started instance(s).

### interval_tasks

This module contains tasks that are executed at intervals, and is imported at the time the server is started. Much of *feederd*'s 'intelligence' can be found here.

All functions prefixed with `task_` are task functions that are registered with the Twisted reactor. All functions prefixed with `threaded_` are the interesting bits that actually do things.

`media_nommer.feederd.interval_tasks.`**register_tasks**()
Registers all tasks. Called by the *feederd* Twisted plugin.

`media_nommer.feederd.interval_tasks.`**task_check_for_job_state_changes**()
Checks for job state changes in a non-blocking manner.

Calls `threaded_check_for_job_state_changes()`.

`media_nommer.feederd.interval_tasks.`**task_manage_ec2_instances**()
Calls the instance creation logic in a non-blocking manner.

Calls `threaded_manage_ec2_instances()`.

`media_nommer.feederd.interval_tasks.`**task_prune_jobs**()
Prune expired or abandoned jobs from the domain specified in the `SIMPLEDB_JOB_STATE_DOMAIN` setting. Also prunes *feederd*'s job cache.

Calls `threaded_prune_jobs()`.

`media_nommer.feederd.interval_tasks.`**threaded_check_for_job_state_changes**()
Checks the SQS queue specified in the `SQS_JOB_STATE_CHANGE_QUEUE_NAME` setting for announcements of state changes from the EC2 instances running *ec2nommerd*. This lets *feederd* know it needs to get updated job details from the SimpleDB domain defined in the `SIMPLEDB_JOB_STATE_DOMAIN` setting.

`media_nommer.feederd.interval_tasks.`**threaded_manage_ec2_instances**()
Looks at the current number of jobs needing encoding and compares them to the pool of currently running EC2 instances. Spawns more instances as needed.

See source of `media_nommer.feederd.ec2_instance_manager.EC2InstanceManager.spawn_if_needed(`
for the logic behind this.

media_nommer.feederd.interval_tasks.**threaded_prune_jobs**()
Sometimes failure happens, but a Nommer doesn't handle said failure gracefully. Instead of state changing to ERROR, it gets stuck in some un-finished state in the SimpleDB domain defined in SIMPLEDB_JOB_STATE_DOMAIN setting.

This process finds jobs that haven't been updated in a very long time (a day or so) that are probably dead. It marks them with an ABANDONED state, letting us know something went really wrong.

### job_cache

Basic job caching module.

**class** media_nommer.feederd.job_cache.**JobCache**
Caches currently active media_nommer.core.job_state_backend.EncodingJob objects. This is presently only un-finished jobs, as defined by media_nommer.core.job_state_backend.JobStateBackend.FINISHED_STATES.

**CACHE = {}**

**classmethod abandon_stale_jobs**()
On rare occasions, nommers crash so hard that no ERROR state change is made, and the job just gets stuck in a permanent unfinished state (DOWNLOADING, ENCODING, UPLOADING, etc). Rather than hang on to these indefinitely, abandon them by setting their state to ABANDONED.

The threshold for which jobs are considered abandoned is configurable via the FEEDERD_ABANDON_INACTIVE_JOBS_THRESH setting.

**classmethod get_cached_jobs**()
Returns a dict of all cached jobs. The keys are unique IDs, the values are the job objects.

**Return type** dict

**Returns** A dictionary with the keys being unique IDs of cached jobs, and the values being EncodingJob instances.

**classmethod get_job**(*job*)
Given a job's unique id, return the job object from the cache.

**Parameters** **job** (EncodingJob) – A job's unique ID or a job object.

**Return type** EncodingJob

**Returns** The cached encoding job.

**classmethod get_jobs_with_state**(*state*)
Given a valid job state (refer to media_nommer.core.job_state_backend.JobStateBackend.JOB_STATES) return all jobs that currently have this state.

**Parameters** **state** (*str*) – The job state to query by.

**Return type** list of EncodingJob

**Returns** A list of jobs matching the given state.

**classmethod is_job_cached**(*job*)
Given a job object or a unique id, return True if said job is cached, and False if not.

**Parameters** **job** (str or EncodingJob) – A job's unique ID or a job object.

**Return type** bool

**Returns** True if the given job exists in the cache, False if otherwise.

**classmethod load_recent_jobs_at_startup()**
Loads all of the un-finished jobs into the job cache. This is performed when *feederd* starts.

**classmethod refresh_jobs_with_state_changes()**
Looks at the state SQS queue specified by the SQS_JOB_STATE_CHANGE_QUEUE_NAME setting and refreshes any jobs that have changed. This simply reloads the job's details from SimpleDB.

**Return type** list of EncodingJob

**Returns** A list of changed EncodingJob objects.

**classmethod remove_job**(*job*)
Removes a job from the cache.

**Parameters** **job** (str or EncodingJob) – A job's unique ID or a job object.

**classmethod uncache_finished_jobs()**
Clears jobs from the cache after they have been finished.

TODO: We'll eventually want to clear jobs from the cache that haven't been accessed by the web API recently.

**classmethod update_job**(*job*)
Updates a job in the cache. Creates the key if it doesn't already exist.

**Parameters** **job** (EncodingJob) – The job to update (or create) a cache entry for.

## 4.3 Indices and tables

- *genindex*
- *modindex*
- *search*

## m

## A



## B



## C



## D



## E



## F



## G

## S



## T



## U



## W