Deme Documentation

Release 0.9git

Mike Mintz

July 17, 2013

CONTENTS

1 Installing Deme **3**
- 1.1 Checking out the source code . . . 3
- 1.2 Required Dependencies . . . 3
- 1.3 Optional Dependencies . . . 5
- 1.4 Setting up Deme . . . 7
- 1.5 Upgrading Deme . . . 7

2 Using Deme **9**
- 2.1 Interface Guide . . . 9
- 2.2 Customizing the Site . . . 9
- 2.3 Advanced Customization . . . 10

3 Advanced Scripts **15**
- 3.1 Creating Users from CSV File . . . 15

4 Deme Architecture **17**
- 4.1 Back-end (items) . . . 17
- 4.2 Front-end (viewers) . . . 31
- 4.3 Modules . . . 33
- 4.4 Email integration . . . 33
- 4.5 Bootstrap Framework . . . 33

5 Code Documentation **35**
- 5.1 Item Types (cms.models) . . . 35
- 5.2 Item Viewers (cms.views) . . . 35

6 Indices and tables **37**

Deme is a tool for making collaborative websites. Contents:

CHAPTER

ONE

INSTALLING DEME

1.1 Checking out the source code

Deme is currently only available for checkout at Github:

http://github.com/mikemintz/deme/tree/master

To install Git and learn how to use it, visit:

http://help.github.com/linux-set-up-git/

To checkout Deme, type:

```
git clone git://github.com/mikemintz/deme.git
```

1.2 Required Dependencies

1.2.1 PostgreSQL

We develop against PostgreSQL from versions 8.3 to 9.2 as our database, although other database engines may be supported. To install PostgreSQL:

- On Ubuntu: `sudo apt-get install postgresql libpq-dev`
- On Mac OS X: Download http://www.enterprisedb.com/getfile.jsp?fileid=484
- Other OS: Download the database at http://www.postgresql.org/download/

You'll also need to configure a user that Django can use to authenticate. If you don't know how to set up users in PostgreSQL, search Google for setting up PostgreSQL and Django on your OS of choice, and find a tutorial.

But here is what I did for reference:

```
$ sudo -u postgres psql template1
template1=# ALTER USER postgres WITH ENCRYPTED PASSWORD 'MYPASSWORD';
template1=# \q
```

I then opened `/etc/postgresql/8.4/main/pg_hba.conf` in a text editor as root. I found the line that said `local all postgres ident`, commented it out, and added a new line underneath that said `local all postgres md5`. I also found the line that said `local all all ident`, commented it out, and added a new line underneath that said `local all all md5`. I also made a `.pgpass` file in my home directory that said `localhost:*:*:postgres:MYPASSWORD` and ran `chmod 600 .pgpass`

1.2.2 Python

We require Python 2.5, 2.6, or 2.7. Python 3 will not work.

- On Ubuntu: `sudo apt-get install python python-dev`
- On Mac OS X: You probably have it already, but if you don't, try http://python.org/ftp/python/2.6.2/python-2.6.2-macosx2009-04-16.dmg
- Other OS: Find out at http://python.org/download/

1.2.3 Virtualenv

We use virtualenv to manage Python libraries.

- On Ubuntu: `sudo apt-get install python-virtualenv`
- On Mac OS X: `sudo easy_install virtualenv`
- Other OS: Find out at http://www.virtualenv.org/

Once you've installed virtualenv, you can install the python dependencies as follows:

```
$ cd /var/www/deme # or wherever you checked it out
$ cd deme_django
$ mkdir venv
$ virtualenv venv
$ source venv/bin/activate
$ pip install Django==1.5.1
$ pip install South==0.8
$ pip install psycopg2
$ pip install django-pure-pagination
```

1.2.4 TinyMCE

We use TinyMCE for WYSIWYG editing in the browser. To install, download the latest package at:

http://tinymce.moxiecode.com/download.php

You'll want to unzip it as `static/javascripts/tinymce`. To ensure you have the paths correct, you should be able to navigate to `static/javascripts/tinymce/jscripts`.

1.2.5 Crystal Project icons

Presently we use Crystal Project icons from everaldo.com in some parts of the UI. To install, download the icons at:

http://www.everaldo.com/crystal/crystal_project.tar.gz

You'll want to unzip it as `static/crystal_project`. To ensure you have the paths correct, you should be able to navigate to `static/crystal_project/16x16`.

1.2.6 Postfix

We use Postfix to deliver notifications and process incoming emails. Other MTAs may work too, but we give configuration instructions for Postfix. To install:

- On Ubuntu: `sudo apt-get install postfix`

- On Mac OS X: You should already have it, but if it's not running you'll want to type `sudo postfix start`
- Other OS: Find out at http://www.postfix.org/

If you want to process incoming mail (optional) in order allow people to reply to action notices and generate comments, you should configure Postfix as follows.

You need to route incoming mail to `script/incoming_email.py`. I have Deme installed at `/var/www/deme/deme_django`, so I added the following to the end of `/etc/postfix/master.cf`:

```
# Deme incoming mail
deme      unix  -       n       n       -       -       pipe
  flags= user=www-data argv=/var/www/deme/deme_django/script/incoming_email.py ${mailbox}
```

I then added the following to the end of `/etc/postfix/main.cf`:

```
# Deme incoming mail
transport_maps = regexp:/etc/postfix/deme_transport
virtual_mailbox_domains = deme.stanford.edu
virtual_mailbox_base = /var/mail
```

You may also have to comment out the line in `main.cf` that starts with `mydestination =`.

I then created a file called `/etc/postfix/deme_transport` containing the following:

```
/.*@deme\.stanford\.edu/    deme:
```

1.3 Optional Dependencies

1.3.1 Python-OpenID

If you want to enable OpenID for authentication, you will have to install the Python OpenID library.

- On Ubuntu: `sudo apt-get install python-openid`
- On Mac OS X: `sudo easy_install python-openid`
- Other OS: Find out at http://openidenabled.com/python-openid/

1.3.2 Graphviz

If you want to generate and display the Deme item type "code graph", you will need to install graphviz.

- On Ubuntu: `sudo apt-get install graphviz`
- Other OS: Find out at http://www.graphviz.org/

1.3.3 Apache

If you want to run Deme in the background all the time (instead of using `./manage.py runserver` to develop), you'll want to set up a server. I chose to use Apache with mod_wsgi, but anything can work.

First, install Apache and mod_wsgi, and make sure mod_wsgi is enabled.

Make sure `DJANGO_SERVES_STATIC_FILES` is false in settings.py to let Apache serve static files.

Here's what I have in my apache `/etc/apache2/sites-available/deme` config file:

```
<VirtualHost *:80>
    ServerName deme.stanford.edu
    ServerAlias deme

    Alias /static /var/www/deme/deme_django
    <Location "/static">
        SetHandler None
    </Location>

    Options -indexes
    RewriteEngine On
    RewriteRule   ^/static/modules/([^/]*)/(.*)  /static/modules/$1/static/$2  [QSA,L,PT]
    RewriteRule   ^/static/(.*)  /static/static/$1  [QSA,L,PT]

    WSGIScriptAlias / /var/www/deme/deme_django/apache/django.wsgi

    BrowserMatch ^Mozilla/4 gzip-only-text/html
    BrowserMatch ^Mozilla/4.0[678] no-gzip
    BrowserMatch bMSIE !no-gzip !gzip-only-text/html
    AddOutputFilterByType DEFLATE text/html text/plain text/css text/xml text/javascript application/
</VirtualHost>

<VirtualHost *:443>
    ServerName deme.stanford.edu
    ServerAlias deme

    SSLEngine On
    SSLCertificateFile /etc/apache2/ssl/server.crt
    SSLCertificateKeyFile /etc/apache2/ssl/server.key

    <Location "/viewing/webauthaccount/login">
        AuthType WebAuth
        Require valid-user
    </Location>

    Alias /static /var/www/deme/deme_django
    <Location "/static">
        SetHandler None
    </Location>

    Options -indexes
    RewriteEngine On
    RewriteRule   ^/static/modules/([^/]*)/(.*)  /static/modules/$1/static/$2  [QSA,L,PT]
    RewriteRule   ^/static/(.*)  /static/static/$1  [QSA,L,PT]

    WSGIScriptAlias / /var/www/deme/deme_django/apache/django.wsgi

    BrowserMatch ^Mozilla/4 gzip-only-text/html
    BrowserMatch ^Mozilla/4.0[678] no-gzip
    BrowserMatch bMSIE !no-gzip !gzip-only-text/html
    AddOutputFilterByType DEFLATE text/html text/plain text/css text/xml text/javascript application/
</VirtualHost>
```

1.3.4 Webauth

If you want to enable Webauth, it's kind of tricky. Here is what I did:

```
sudo apt-get install libapache2-webauth
sudo a2enmod webauth
sudo mkdir -p /etc/apache2/conf/webauth
ssh mintz@pod.stanford.edu "wallet -f keytab get keytab webauth/deme.stanford.edu"
sudo scp mintz@pod.stanford.edu:keytab /etc/apache2/conf/webauth/keytab
sudo chown root:www-data /etc/apache2/conf/webauth/keytab
sudo chmod 640 /etc/apache2/conf/webauth/keytab
ssh mintz@pod.stanford.edu "rm keytab"
sudo apt-get install krb5-user
sudo scp mintz@pod.stanford.edu:/usr/pubsw/etc/krb5.conf /etc/krb5.conf
```

You also need SSL working, which you can figure out from here on Ubuntu: http://www.tc.umn.edu/~brams006/selfsign.html http://www.tc.umn.edu/~brams006/selfsign_ubuntu.html `sudo apt-get install ca-certificates`

Add this to the bottom of `/etc/apache2/apache2.conf`: WebAuthKeyring conf/webauth/keyring WebAuthKeytab conf/webauth/keytab WebAuthServiceTokenCache conf/webauth/service_token_cache WebAuthLoginURL https://weblogin.stanford.edu/login/ WebAuthWebKdcURL https://weblogin.stanford.edu/webkdc-service/ WebAuthWebKdcPrincipal service/webkdc@stanford.edu WebAuthSSLRedirect on

1.4 Setting up Deme

To set up Deme, you first must copy `settings.py_EXAMPLE` to `settings.py`. Edit `settings.py` and make sure the database username/password is correct, and generate a random `SECRET_KEY`. You'll want to set `DEFAULT_HOSTNAME` and `NOTIFICATION_EMAIL_HOSTNAME` accordingly for outgoing emails.

If you are using PostgreSQL with username `postgres` and database `deme_django`, you can quickly initialize the database by running:

```
script/reset_db.sh
```

If you want something to be different, just customize this file and run it with your own settings.

You can re-run this file every time you want to reset everything.

To see if everything is working, run:

```
./manage.py runserver
```

And visit http://localhost:8000/ on your computer. With any luck, Deme will be working!

1.5 Upgrading Deme

When you upgrade Deme (by issuing a new `git pull`), you should also type `./manage.py migrate` to make any necessary updates to the database schema.

CHAPTER

TWO

USING DEME

Deme is designed for the Chrome, Safari, and Firefox browsers. It should also work in Internet Explore 8 and above, but will have visual inconsistencies.

2.1 Interface Guide

At the top is the Toolbar, which allows users to Log in, Search, etc. By default, the Toolbar is in "Basic Layout", which hides the more advanced functionality. Clicking on the Gear icon at the very top left switches to "Advanced Layout", which introduces more advanced funtionality.

2.1.1 Basic Layout

Buttons from left to right:

Gear: Switches between Basic and Advanced Layout. Hidden if the user does not have permission to view the Advanced Layout. **Font size:** Allows the user to change the font size for all text on the screen. **Search:** Searches all items by name and provides a quick link to the Advanced Search interface. **Login menu:** Displays log in options as well as details for logged in users.

2.1.2 Advanced Layout

Buttons from left to right not present in the Basic Layout:

Breadcrumbs: A list showing the current item (if any), item type, and a link to all items. **Actions menu:** A menu of actions that apply to the current page. **Actions icons:** Shortcuts to some of the key actions of the Actions menu. **Metadata:** Opens the Metadata sidebar, which contains help and information about the current page.

2.2 Customizing the Site

After your Deme site is appropriately set up, many components can be easily customized. Be sure to log in as Admin in order to have permission to make many of these changes.

2.2.1 Change the Site Title and Logo

1. Go to your current site's settings. By default, your site will be `Default Site`. Either look for `Default Site` using search or, in the footer, click `Admin > Sites > Default Site`.

2. Your site settings should show that *Site logo* is "None" and *Site title* is blank.
3. Go to `Actions > Edit` to change them.

Tips:

- To add an image logo, click the ... button to choose or upload an image. To upload a new image, click on `Create a new image document`, then give your image a name, select your image file to upload, and create the image. The popup window will then close and your image's name will show up in the image logo field. Be sure to save for the logo to show up.
- To edit the title, simply add a title and save.

2.2.2 Edit the Home Page

1. From the `Home Page`, click the `Edit` button
2. The editor uses TinyMCE as a WYSIWYG editor. After editing, click `Save HTML document` to commit your changes.

2.3 Advanced Customization

2.3.1 Hiding *Advanced Layout*

To change who can see the Advanced toolbar, you'll need to change *Advanced Layout* under *Global Permissions*:

1. Log in as Admin by clicking on the person icon in the top right of the site, then clicking *Login as* and choosing *Admin*
2. Go to the site Admin page by clicking its link in the footer
3. Go to Global Permissions
4. Under *Everyone*, remove the *Advanced Layout* permission by clicking its [-] button and then save. Now the only user who sees the Advanced Layout bar is the Admin.
5. Let's say we want logged in users to see the Advanced Layout bar but not Anonymous users. First, let's allow all users to see the bar by reenabling *Advanced Layout* under *Everyone* by clicking [+] and saving. Now, let's click *Assign a Permission to a User* at the very bottom of the permissions and enter *Anonymous* and then click *Add Collection*. Next, in the newly generated permission area for Anonymous, add a *New Permission*, from its dropdwon menu select *Advanced Layout*, click [-] to disallow, and then finally *Save Permissions*.
6. Now if you visit the site as an Anonymous user, you'll no longer see the Advanced Layout tools.

2.3.2 Understanding DjangoTemplateDocuments

DjangoTemplateDocuments allow many different sections of the page to be modified. Here are the important ones:

- `{% block content %}...{% endblock %}` addresses the main content area of a page
- `{% block banner-section %}...{% endblock %}` addresses the area above the main content area
- `{% block tabs-section %}...{% endblock %}` addresses the area between the content and banner-section areas
- `{% block footer %}...{% endblock %}` addresses the footer area
- `{% block logo-section %}...{% endblock %}` addresses the logo area

- `{% block sidebar-section %}...{% endblock %}` addresses the sidebar underneath the logo
- `{% block custom_css %}...{% endblock %}` allows insertion of custom CSS styles

To customize your site, you edit these different sections. To learn more about the syntax and technical details, visit [Django's Templates Documentation](https://docs.djangoproject.com/en/1.5/ref/templates/)

2.3.3 Switching the Site Layout

1. Go to your current site's settings. By default, your site will be `Default Site`. Either look for `Default Site` using search or, in the footer, click `Admin > Sites > Default Site`.
2. The `Default layout` is used on all pages of the current site. By default, it should be `Deme Layout`. Click on it to open it for editing.

- Deme also comes with some sample layouts. By default they are called "Sample Layout" and "Sample Forum Layout". Change the `Default layout` to one of them for a very different look and feel.

2.3.4 Editing the Site Layout

In the current layout, edit the different "blocks" to modify the site.

Adding navigation tabs:

```
{% block tabs-section %}
<div class="tabs-section">
  <ul class="nav nav-tabs">
    <li><a href="/">Home</a></li>
    <li><a href="/viewing/group/5">Group</a></li>
    <li><a href="...">...</a></li>
  </ul>
</div>
{% endblock %}
```

Inserting a banner section:

```
{% block banner-section %}
<div class="banner-section">
  <h3>Your banner text goes here</h3>
</div>
{% endblock banner-section %}
```

Adding custom CSS:

```
{% block custom_css %}
.page-layout .logo-section a.logo {
  background: darkred;
}
{% endblock %}
```

Adding CSS/JS files:

```
{% block head_append %}
<link rel="stylesheet" href="http://www.yoursite.com/stylesheet.css" type="text/css">
{% endblock %}
```

Editing the sidebar:

```
{% block sidebar-section %}
<div class="sidebar-section">
  <div class="panel">
    <div class="panel-heading">
      Resources
    </div>
    <ul>
        <li><a href="...">Link Goes Here</a></li>
        <li>...</li>
    </ul>
  </div>
</div>
{% endblock %}
```

Showing a different footer to users who aren't the Admin:

```
{% block footer %}
  {% if cur_agent.is_admin %}
    {{ block.super }}
  {% else %}
    This is the footer you see when you're not the admin.
  {% endif %}
{% endblock footer %}
```

Showing a log in form to visitors who aren't logged in:

```
{% block body_wrap %}
  {% if cur_agent.is_anonymous %}
    {% include "demeaccount/required_login_include.html" %}
  {% else %}
    {{ block.super }}
  {% endif %}
{% endblock body_wrap %}
```

Hiding chat:

```
{% block chat %}{% endblock %}
```

To show site-wide chat, you'd simply delete that line from a layout.

2.3.5 Using a DjangoTemplateDocument as the Home Page (Advanced)

Assuming you've created a DjangoTemplateDocument item you'd like to use as your home page:

1. Go to your current site's settings. By default, your site will be `Default Site`. Either look for `Default Site` using search or, in the footer, click `Admin > Sites > Default Site`.
2. Edit `Aliased item` to be the desired item. Change `Viewer` to "djangotemplatedocument" and `Action` to "render".

Using a DjangoTemplateDocument instead of an HtmlDocument allows editing of nearly all elements on a page instead of only the contents of the main content area.

2.4 Using Deme with Multiple Sites

2.4.1 Using a single Deme installation with multiple domains/subdomains

If you have multiple domains or subdomains that point to your Deme installation, you can show different sites for each by using different Site objects. For instance, let's say you are pointing both *domain1.com* and *domain2.com* to the same Deme installation. Let's also say that you've already set up domain1.com as desired but want *domain2.com* to appear differently.

1. List all sites by, while logged in as Admin, clicking Admin in the footer, then All Sites. There should be one site, the default Site that shows whenever a subdomain doesn't match a particular Site.
2. From here, click **Create a new site**.
3. Set up the site to work with your domain. To do this, change the `Hostname` to "domain2.com".
4. Now, the two sites should be using different Sites. Setting the `Site title` and `Default layout` are simple ways to check to make sure the sites are showing different content.

2.4.2 Restricting items to a single group

If you have a lot of different sites, you may wish to restrict certain items to certain groups. To do so:

1. From the item you want restricted, go to the Actions menu, then Modify Permissions.
2. Under *Everyone*, add a New Permission. Choose *View Anything* and click on the **[-]** negative permission. This make it so that no one can view the document.
3. Click **Assign a Permission to a Group of Users**. Select the group you'd like to be able to see the document.
4. Under your group, add a New Permission. Choose *View Anything* and click on the **[+]** positive permission. This makes it so that your group can view the document.

CHAPTER

THREE

ADVANCED SCRIPTS

3.1 Creating Users from CSV File

The script file is located at /script/add_agents_from_csv.py. To use it, upload a CSV file to the server and invoke the script with the CSV file as argument.

The CSV file should be formatted "name", "password", "group". The group must be created ahead of time. For example, to add users to the group "Deliberation Group Alpha", the CSV file might look like:

```
Mike,secretpassword,Deliberation Group Alpha
Todd,anothersecretpassword,Deliberation Group Alpha
Michael,differentpassword,Deliberation Group Alpha
```

Then each user would be able to login with their name and password.

CHAPTER

FOUR

DEME ARCHITECTURE

4.1 Back-end (items)

4.1.1 Overview

Most persistent data in Deme is stored in "items". An item is an instance of a particular "item type". This is in parallel to object-oriented programming where instances (items) are defined by classes (item types), and in parallel to filesystems where files (items) are defined by file types (item types). The Deme item types form a hierarchy through inheritance, so if the Person item type inherits from the Agent item type, then any item that is a person is also an agent. Every item type inherits from the Item item type (which corresponds to the Object class in many programming languages). We allow multiple inheritance, and use it occasionally (e.g., TextComment inherits from both Comment and TextDocument).

4.1.2 Database

We use ORM with multi-table inheritance. There is a database table for every item type, and a row in that table for every item of that item type. For example, if our entire item type hierarchy is Item -> Agent -> Person, and our items are Mike[Person] and Robot[Agent], then there will be one row in the Person table (for Mike), two rows in the Agent table (for Mike and Robot), and two rows in the Item table (for Mike and Robot).

4.1.3 Fields

Every item type defines the "fields" relevant for its items, and item types inherit fields from their parents. As a simple example, imagine Item defines the "description" field, Agent defines no new fields, and Person defines the "first_name" field. Therefore every person has a description and a first_name. The columns in each table correspond to the fields in its item type. So if the Mike item has description="a programmer" and first_name="Mike", then his row in the Item table will just have description="a programmer", his row in the Agent table will have no fields (because Agent did not define any new fields), and his row in the Person table will have first_name="Mike".

4.1.4 Field types

Every field has a type that corresponds to the types we can store in our database. The basic types are things like String, Integer, and Boolean. It is important to realize that fields are *not* items. So if Mike's first_name field is of type String, it cannot be referred to as an item itself. You cannot store entire items as fields, but you can have fields that point to other items (foreign keys in database-speak, pointers/references in programming, links in filesystems). If we wanted to "itemize" the first_name field, we could make a new FirstName item type and have the Person's first_name field be a pointer to an first_name item. In the case of first_name, however, this is not particularly useful, and it just

adds more overhead (and makes versioning difficult, as we'll see later on). Pointer fields are more useful for defining relationships between legitimate items. For example, the Item item type has an "creator" field pointing to the agent that created the item.

Pointers do not represent an exclusive "ownership" relationship. I.e., just because an item pointed to the agent that wrote it, this does not prevent other items from pointing to that agent. Multiple items can point to a common item.

Fields cannot store data structures like lists. If you want to express X has many Y's, rather than storing all the Y's in the X row, you should itemize the Y's, and have each Y point to the X that it belongs to. For example, an Agent has many ContactMethods. So rather than storing the ContactMethods as fields inside each Agent, we make ContactMethod an item type, and give it an "Agent pointer" field. So the contact methods for agent 123 are represented by all of the ContactMethods that have agent_pointer=123.

The most important field is the id field (primary key in database-speak, memory address of the object in programming, inode number in filesystems). Every item has a unique id, an auto-incrementing integer starting at 1. Items share the same id with their parent-item-type versions (so Mike's row in the Person table has the same id as Mike's row in the Agent table and Item table). Pointer fields are effectively references to the id of the pointee. It is important that the id field never change so that there is always a single reliable way to refer to a particular item. No other field is guaranteed to be unique among all items (although some item types define unique fields within that item type, such as DemeAccount's unique username).

Some fields are specified as immutable, which means once they are set, they cannot be changed. The id field is a prime example of an immutable fields, but other fields like creator and created_at are immutable as well.

4.1.5 Versioning

For every item type table, there is a dual "revisions" table. So in our previous example, in addition to the Item, Agent, and Person tables, we now have ItemVersion, AgentVersion, and PersonVersion tables. These tables store the exact same fields as their original non-versioned tables, with a few exceptions:

- Each itemversion has a pointer to the item it is a version of.
- Each itemversion has a version_number field, an integer starting at 1 and incrementing each time a new version is made. The largest version number always corresponds to the latest version.
- Immutable fields are not stored in the version table. For example, although creator and created_at are fields in the Item table, they are not fields in the ItemVersion table since they cannot change.
- Uniqueness constraints are not propagated from the regular table to the version table. For example, if we had the constraint that there can only be one Person for every email address (i.e., email addresses are unique), then it wouldn't make sense to enforce that in the revisions table, since version 1 and version 2 of Mike might have the same email address.

So apart from these differences, each itemversion stores the exact same fields as the regular item. Every time a change is made to the regular item, a snapshot is taken and a new itemversion is created with an increased version number. So when the Mike item is changed, Mike's rows in the Item, Agent, and Person table are updated, and those updates are copied over to the ItemVersion, AgentVersion, and PersonVersion tables, so that we can refer to the past.

Here is the major caveat. Imagine there is a student and a class. We must represent their relationship with a third item type, the ClassMembership, since classes cannot store arrays of students, and students cannot store arrays of classes. If the student joins the class, a ClassMembership is created. Ideally, we'd like to be look at the previous roster of the class, but since the class roster is just composed of ClassMemberships that point to the class, there is no way to refer to a previous version. A possible solution is to refer to the entire state of items at a particular time step, which is possible since we can compute what versions were around at any given moment, but that gets convoluted. For now, just assume that versioning only plays well with regular fields, and does not work on data structures created via relationship tables.

4.1.6 Deleting items

There are two ways of deleting items: deactivating and destroying. Neither of these methods removes any rows from the database. Deactivating is recoverable (by reactivating), destroying is not. The user interface ensures that deactivating happens before destroying.

- **Deactivating:** If an agent deactivates an item, it sets the active field to false. An agent can recover the item by reactivating it, which sets the active field back to true. Each time this happens, the version does not change, but a DeactivateActionNotice or ReactivateActionNotice is automatically generated to log when the item was active. Inactive items can still be viewed and edited as normally. The major difference between an active and an inactive item is that when an item is inactive, it will not be returned as the result of queries (unless the query specifically requests inactive items). For example, when you look at the list of students in a class, it will only show students that are active with classmemberships that are active.
- **Destroying:** After an item is deactivated, you can permanently nullify all of its fields (and/or the fields in its versions) so that it is impossible to recover (but keep active=false). A DestroyActionNotice is automatically generated to log when the item was around.

  Our solution is as follows. We allow any field to have the special NULL value from SQL. The application (not the database) ensures that fields only take on these values when the item is destroyed, and never otherwise (I haven't finished making sure this happens yet). Thus, to destroy an item is to set every field to NULL, and set destroyed=True (and leave alone id, item_type, and active, version_number). Destroying an item also removes all permissions and versions of the item. After an item is destroyed, nobody can make changes (in particular, it cannot be reactivated or edited).

  Normally, having NULL values makes the code much more complex and prone to bugs, since the developer has to write a lot of checks for NULL. For example, to display the name of the creator of an item, the developer would have to write something like `if (item.creator != NULL && item.creator.name != NULL) ...`. Since we already do all of this up-front error checking in the permission system (to ensure that the logged in agent has permission to view the creator of the item and the name of the creator), all we have to do is modify the permission code so that users cannot view fields (or take any actions) for destroyed items. So if an item's creator was destroyed, a simple viewer will just display the creator's name in the same way it would display something it does not have permission to view (a more advanced viewer could check to see if it was destroyed).

  It will also be possible to destroy specific versions of an item (not yet implemented). You can destroy any version except for the latest version (if you want to destroy the latest version, just edit the item to make a new version so that the version you want to destroy is now the second-latest). Destroying a version will permanently NULLify all fields in the version.

4.1.7 Things stored outside the database

Not every bit of persistent data is stored in the database in item fields. Here are the exceptions so far:

- Uploaded files (like the files corresponding to FileDocuments) are stored on the filesystem in the static files folder so they can be stored more efficiently (databases are not good for binary data) and so they can be served quickly by the webserver without going through Deme. The FileDocument item type has a string field that represents to the path on the filesystem to the file.
- Item type definitions are stored as code, not in the database. The fact that Person is a subtype of Agent and defines the first_name field is inferred from the Deme code, and should not be read from the database. In the future, we are considering creating a "ItemTypes" table that stores one row per item type (the size would remain fixed as long as the code does not change), and this way, we could refer to item types (one good example is an admin might want to create a permission for another user to create new items of a specified type). This would also be a good place to store dynamic settings specific to each item type (like default permissions). Since the item type definitions are static, it seems like we never need this ability, and can always emulate it with more code.

4.1.8 Core item types

Below are the core item types and the role they play (see the full ontology at http://deme.stanford.edu/viewing/codegraph).

- **Item:** Item is item type that everything inherits from. It gives us a completely unique id across all items. It defines two user-editable fields (`name` and `description`) and six automatically generated fields (`id`, `version_number`, `item_type`, `creator`, `created_at`, `active`, and `destroyed`).
  - The `name` field is the friendly name to refer to the specific item: the title of a document or the preferred name of a person, and is the kind of name that would appear as the <title> of a webpage or the text of a link to that item. Currently, the name field cannot be blank (so that the viewer always has some text to display), but we are considering making it blank for items that don't need names (like Memberships) and having the viewer deal with possibly blank names.
  - The `description` field is a string field for metadata, which can be used for any purpose. Generally, the description is not considered part of the body of the item itself, but tells what the item is. The description for a budget document item might read, "This is the budget as drafted by the budget committee."
  - The `id` field is an automatically incrementing integer that gives a globally unique identifier for every item.
  - The `version_number` field is the latest version number.
  - The `item_type` field is the name of the actual item type at the lowest level in the inheritance graph.
  - The `creator` field is a pointer to the Agent that created the item.
  - The `created_at` field is the date and time the item was created.
  - The `active` field is true or false, depending on whether the item is active or not.
  - The `destroyed` field is true or false, depending on whether the item is destroyed or not.
  - The `default_viewer` field is the name of the default viewer to display this item (for the various places Deme generates links to an item).
  - The `email_list_address` field optionally specifies a custom email address (the portion before the at-sign) when emails are sent to the subscribers of this item.
  - The `email_list_subject` field optionally specifies a prefix of the subject field of emails sent to subscribers of this item.
  - The `email_sets_reply_to_all_subscribers` field defines whether the reply-to field of emails sent to subscribers of the item goes to the subscribers (if True) or the sender (if False).
- **Webpage:** A Webpage is an item with one extra field, `URL`, used to represent external webpages.

Agents and related item types

- **Agent:** This item type represents an agent that can "do" things. Often this will be a person (see the Person subclass), but actions can also be performed by other agents, such as bots and anonymous agents. Agents are unique in the following ways:
  - Agents can be assigned permissions
  - Agents show up in the creator and updater fields of other items
  - Agents can authenticate with Deme using AuthenticationMethods
  - Agents can be contacted via their ContactMethods
  - Agents can subscribe to other items with Subscriptions

  There is only one field defined by this item type, `last_online_at`, which stores the date and time when the agent last accessed a viewer.

- **AnonymousAgent:** This item type is the agent that users of Deme authenticate as by default. Because every action must be associated with a responsible Agent (e.g., updating an item), we require that users are authenticated as some Agent at all times. So if a user never bothers logging in at the website, they will automatically be logged in as an AnonymousAgent, even if the website says "not logged in". There should be exactly one AnonymousAgent at all times.

  This item type does not define any new fields.

- **GroupAgent:** This item type is an Agent that acts on behalf of an entire group. It can't do anything that other agents can't do. Its significance is just symbolic: by being associated with a group, the actions taken by the group agent are seen as collective action of the group members. In general, permission to login_as the group agent will be limited to powerful members of the group. There should be exactly one GroupAgent for every group.

  This item type defines one field, a unique `group` pointer that points to the group it represents.

- **AuthenticationMethod:** This item type represents an Agent's credentials to login. For example, there might be a AuthenticationMethod representing my Facebook account, a AuthenticationMethod representing my WebAuth account, and a AuthenticationMethod representing my OpenID account. Rather than storing the login credentials directly in a particular Agent, we allow agents to have multiple authentication methods, so that they can login different ways. In theory, AuthenticationMethods can also be used to sync profile information through APIs. There are subclasses of AuthenticationMethod for each different way of authenticating.

  This item type defines one field, an `agent` pointer that points to the agent that is holds this authentication method.

- **DemeAccount:** This is an AuthenticationMethod that allows a user to log on with a username and a password. The username must be unique across the entire Deme installation. The password field is formatted the same as in the User model of the Django admin app (algo$salt$hash), and is thus not stored in plain text.

  This item type defines four fields: `username`, `password`, `password_question`, and `password_answer` (the last two can be used to reset the password and send it to the Agent via one of its ContactMethods).

- **Person:** A Person is an Agent that represents a person in real life. It defines four user-editable fields about the person's name: `first_name`, `middle_names`, `last_name`, and `suffix`.

- **ContactMethod:** A ContactMethod belongs to an Agent and contains details on how to contact them. ContactMethod is meant to be abstract, so developers should always create subclasses rather than creating raw ContactMethods.

  This item type defines one field, an `agent` pointer that points to the agent that is holds this contact method.

  Currently, the following concrete subclasses of ContactMethod are defined (with the fields in parentheses):

  - `EmailContactMethod(email)`
  - `PhoneContactMethod(phone)`
  - `FaxContactMethod(fax)`
  - `WebsiteContactMethod(url)`
  - `AIMContactMethod(screen_name)`
  - `AddressContactMethod(street1, street2, city, state, country, zip)`

- **Subscription:** A Subscription is a relationship between an Item and a ContactMethod, indicating that all action notices on the item should be sent to the contact method as notifications. This item type defines the following fields:

  - The `contact_method` field is a pointer to the ContactMethod that is subscribed with this Subscription.
  - The `item` field is a pointer to the Item that is subscribed to with this Subscription.

  - The `deep` field is a boolean, such that when deep=true and the item is a Collection, all action notices on all items in the collection (direct or indirect) will be sent in addition to action notices on the collection itself.

Collections and related item types

- **Collection:** A Collection is an Item that represents an unordered set of other items. Collections just use pointers from Memberships to represent their contents, so multiple Collections can point to the same contained items. Since Collections are just pointed to, they do not define any new fields.

  Collections "directly" contain items via Memberships, but they also "indirectly" contain items via chained Memberships. If Collection 1 directly contains Collection 2 which directly contains Item 3, then Collection 1 indirectly contains Item 3, even though there may be no explicit Membership item specifying the indirect relationship between Collection 1 and Item 3. (In the actual implementation, a special database table called RecursiveMembership is used to store all indirect membership tuples, but it does not inherit from Item.)

  It is possible for there to be circular memberships. Collection 1 might contain Collection 2 and Collection 2 might contain Collection 1. This will not cause any errors: it simply means that Collection 1 indirectly contains itself. It is even possible that Collection 1 *directly* contains itself via a Membership to itself.

- **Group:** A group is a collection of Agents. A group has a folio that is used for collaboration among members. THis item type does not define any new fields, since it just inherits from Collection and is pointed to by Folio.

- **Folio:** A folio is a special collection that belongs to a group. It has one field, the `group` pointer, which must be unique (no two folios can share a group).

- **Membership:** A Membership is a relationship between a collection and one of its items. It defines two basic fields, an `item` pointer and a `collection` pointer. It also defines a `permission_enabled` boolean, which allows permissions to propagate through the containing collection to the member item (explained more in the Permissions section).

Documents

- **Document:** A Document is an Item that is meant can be a unit of collaborative work. Document is meant to be abstract, so developers should always create subclasses rather than creating raw Documents. This item type does not define any fields.

- **TextDocument:** A TextDocument is a Document that has a body that stores arbitrary text. This item type defines one field, `body`, which is a free-form text field.

- **DjangoTemplateDocument:** This item type is a TextDocument that stores Django template code. It can display a fully customized page on Deme. This is primarily useful for customizing the layout of some or all pages, but it can also be used to make pages that can display content not possible in other Documents. This item type defines two new fields:

  - The `layout` field a pointer to another DjangoTemplateDocument that specifies the layout this template should be rendered in (i.e., this template inherits from the layout template in the Django templating system). This field can be null.

  - The `override_default_layout` field is a boolean specifying the behavior when the `layout` field is null. If this field is true and `layout` is null, this template will be rendered without inheriting from any other. If this field is false and `layout` is null, then this field will inherit from the default layout (which is defined by the current Site).

- **HtmlDocument:** An HtmlDocument is a TextDocument that renders its body as HTML. It uses the same `body` field as TextDocument, so it does not define any new fields.

- **FileDocument:** A FileDocument is a Document that stores a file on the filesystem (could be an MP3 or a Microsoft Word Document). It is intended for all binary data, which does not belong in a TextDocument (even though it is technically possible). Subclasses of FileDocument may be able to understand various file formats

and add metadata and extra functionality. This item type defines one new field, `datafile`, which represents the path on the server's filesystem to the actual file.

Annotations (Transclusions, Comments, and Excerpts)

- **Transclusion:** A Transclusion is an embedded reference from a location in a specific version of a TextDocument to another Item. This item type defines the following fields:
  - The `from_item` field is a pointer to the TextDocument that is transcluding the other item.
  - The `from_item_version_number` field is the version number of the TextDocument in which this Transclusion occurs.
  - The `from_item_index` field is a character offset into the body of the TextDocument where the transclusion occurs.
  - The `to_item` field is a pointer to the Item that is referenced by this Transclusion.
- **Comment:** A Comment is a unit of discussion about an Item. Each comment specifies the commented item and version number (in the `item` and `item_version_number` fields). Comment is meant to be abstract, so developers should always create subclasses rather than creating raw Comments. Currently, users can only create TextComments.

  If somebody creates Item 1, someone creates Comment 2 about Item 2, and someone responds to Comment 2 with Comment 3, then one would say that Comment 3 is a *direct* comment on Comment 2, and Comment 3 is an *indirect* comment on Item 1. The Comment item type only stores information about direct comments, but behind the scenes, the RecursiveComment table (which does not inherit from Item) keeps track of all of the indirect commenting so that viewers can efficiently render entire threads.

  A Comment also specifies a `from_contact_method` field, which points to a ContactMethod that was used to generate this comment. Often this will be null, but in cases where people send emails to generate comments, this will point to the EmailContactMethod, and is used to set an appropriate reply address.
- **TextComment:** A TextComment is a Comment and a TextDocument combined. It is currently the only form of user-generated comments. It defines no new fields.
- **Excerpt:** An Excerpt is an Item that refers to a portion of another Item (or an external resource, such as a webpage). Excerpt is meant to be abstract, so developers should always create subclasses rather than creating raw Excerpts.
- **TextDocumentExcerpt:** A TextDocumentExcerpt refers to a contiguous region of text in a version of another TextDocument in Deme. The body field contains the excerpted region, and the following fields are introduced:
  - The `text_document` field is a pointer to the TextDocument being excerpted.
  - The `text_document_version_number` field is the version number of the TextDocument being excerpted.
  - The `start_index` field identifies the character position of the beginning of the region.
  - The `length` field identifies the length in characters of the region.

Viewer aliases

In order to allow vanity URLs (i.e., things other than `/viewing/item/5`), we have a system of hierarchical URLs. In the future, we'll need to make sure URL aliases cannot start with /viewing/ (our base URL for viewers), /static/ (our base URL for static content like stylesheets), or /meta/ (our base URL for Deme framework things like authentication). Right now, if someone makes a vanity URL with one of those prefixes, you just cannot reach it (it does not shadow the important URLs).

- **ViewerRequest:** A ViewerRequest represents a particular action at a particular viewer (basically a URL, although its stored more explicitly). A ViewerRequest is supposed to be abstract, so users can only create Sites and CustomUrls. It specifies the following fields

  - A `viewer` (just a string, since viewers are not Items)
  - An `action` (like "view" or "edit")
  - An `item` that is referred to (or null for item type actions like "list" and "new")
  - A `query_string` if you want to pass parameters to the viewer
  - A `format` (like "html" or "json", for the viewer to know what output to render)

- **Site:** A Site is a ViewerRequest that represents a logical website with URLs. Multiple Sites on the same Deme installation share the same Items with the same unique ids, but they resolve URLs differently so each Site can have a different page for /mike. If you go to the base URL of a site (like http://example.com/), you see the ViewerRequest that this Site inherits from. This item type specifies the following fields:
  - The `hostname` field specifies the hostname of this site, so that the viewer can determine which site a visitor is currently at from the URL.
  - The `default_layout` field is a pointer to a DjangoTemplateDocument. Whenever a visitor is at a URL designated for this site, the template will be rendered under this layout. If this field is null, the Deme default layout (in `cms/templates/default_layout.html`) will be used.
- **CustomUrl:** A CustomUrl is a ViewerRequest that represents a specific path.

  Each CustomUrl has a `parent_url` field pointing to the parent ViewerRequest (it will be the Site if this CustomUrl is the first path component) and a `path` field. So when a user visits http://example.com/abc/def, Deme looks for a CustomUrl with name "def" with a parent with name "abc" with a parent Site with hostname "example.com". In other words, we need to find something that looks like this:

```
CustomUrl(name="def", parent_url=CustomUrl(name="abc", parent_url=Site(hostname="example.com")))
```

Misc item types

- **DemeSetting:** This item type stores global settings for the Deme installation. Each DemeSetting has a unique `key` field and an arbitrary `value` field. Since values are strings of limited size, settings that involve a lot of text (e.g., a default layout) should have a value pointing to an item that contains the data (e.g., the id of a document).

4.1.9 ActionNotices

ActionNotices keep records of every action that occurs in Deme. ActionNotices are not items themselves, but they exist in the database and point to items.

Every ActionNotice keeps the following fields

- Action item (the item that was acted upon)
- Action item version number (the version of the item after the action took place)
- Action agent (the agent who acted upon the item)
- Action time (the date/time that the action took place)
- Action summary (the optional user-entered description of the action – for edits, this is like an "Edit Summary", but it applies to any action)

There are currently 6 types of ActionNotices: DeactivateActionNotices, ReactivateActionNotices, DestroyActionNotices, CreateActionNotices, EditActionNotices, and RelationActionNotices. The first 5 are self-explanatory: when an agent deactivates, reactivates, destroys, creates, or edits an item, this automatically generates an ActionNotice. None of these 5 ActionNotices define new fields. Although it seems like the CreateActionNotices and EditActionNotices should define fields to specify what changed, this information can be inferred from the item itself (and its revisions).

RelationActionNotices are more interesting: when an agent modifies an item (the *from* item) that points to another item (the *to* item), a RelationActionNotice is generated about the *to* item. These notices are only generated when the pointer

changes, either from something else to the *to* item, or from the *to* item to something else. RelationActionNotices define new fields to specify the *from* item and its version at the time of the action, and the field in the *from* item that points to the *to* item.

A good example of a RelationActionNotice is a membership that points to a collection. If I'm viewing the ActionNotices for the collection, I will see a RelationActionNotice saying that at some date, some user set the membership to point to this collection. Or in other words, an item was added to this collection.

In order to view ActionNotices, an agent must have the `view action_notices` permission with respect to the action item. For RelationActionNotices, an agent must also have permission to view the pointing field in the *from* item.

If you are subscribed to an item (via the Subscription item type), and you have permission to view ActionNotices on that item, you will receive notifications by email every time an ActionNotice is generated.

The ActionNotices about an agent include ActionNotices whose `action_agent` field points to the agent, in addition to ActionNotices whose `action_item` field points to the agent. Thus, if you subscribe to an agent, you will get emails about things they do, in addition to things done to them. For this reason, RelationActionNotices are not generated for the `action_agent` field of an item, or else there would be redundant ActionNotices on the same item.

4.1.10 Permissions

Permissions define what actions Agents can and cannot do. Similar to ActionNotices, permissions are not items themselves, but they exist in the database and point to items (it used to be that permissions were items, but for simplicity and efficiency, we now keep them separate).

There are 9 types of permissions, divided among 2 axis: the `source` axis and the `to` axis. Along the `source` axis, permissions can be given at 3 levels: to a single Agent, to the members of a Collection of Agents, or to all Agents. Along the `to` axis, permissions can be applied to 3 levels: to a single Item, to the members of a Collection of Items, or to all Items. For both axes, we refer to these three levels as "one", "some", and "all". The 9 possible permissions are shown in the table below:

.

		To		
		One	Some	All
From	One	OneToOnePermission (1)	OneToSomePermission (2)	OneToAllPermission (3)
	Some	SomeToOnePermission (4)	SomeToSomePermission (5)	SomeToAllPermission (6)
	All	AllToOnePermission (7)	AllToSomePermission (8)	AllToAllPermission (9)

Although we could accomplish anything using only OneToOnePermissions, the other permission types allow us to more concisely express permissions. For example, if our site was a wiki and we wanted any user to be able to edit any document, we would create a single AllToAllPermission, rather than a new OneToOnePermission for every Agent/Item pair.

Each permission, in addition to specifying the `source` and the `to` axes, specifies an `ability` string and an `is_allowed` boolean. When there are multiple permissions with the same `ability`, the permissions at a level with a lower number (shown in parentheses after each permission type in the table above) take precedence. When there are multiple permissions at the same level, the negative (`is_allowed=False`) permissions take precedence over the positive permissions.

On both axes, when we refer to all agents or items in a collection (i.e., [X]ToSome or SomeTo[X]), we refer to both direct and indirect members. Thus, the permission code checks the RecursiveMembership table to determine whether an agent or an item is affected by the permission.

There are two types of abilities: item abilities and global abilities. Item abilities can apply to a particular item (or collection of items), such as "can edit the name of the item"; while global cannot apply to any particular item, such as "can create new documents". Each item type defines the item abilities that are relevant to it, and the global abilities it introduces.

An agent has an ability if there exists a relevant permission with `is_allowed=True` at some level without any relevant permissions with `is_allowed=False` at any levels with the same or lower number.

Below is a list of all possible global abilities:

- `create AIMContactMethod`
- `create AddressContactMethod`
- `create Agent`
- `create Collection`
- `create CustomUrl`
- `create DemeAccount`
- `create DjangoTemplateDocument`
- `create EmailContactMethod`
- `create Event`
- `create FaxContactMethod`
- `create FileDocument`
- `create Group`
- `create HtmlAdvertisement`
- `create HtmlDocument`
- `create ImageDocument`
- `create Membership`
- `create Person`
- `create PhoneContactMethod`
- `create Subscription`
- `create TextAdvertisement`
- `create TextComment`
- `create TextDocument`
- `create TextDocumentExcerpt`
- `create Transclusion`
- `do_anything` (Agents with this ability automatically have every single global ability and every item ability with respect to every item. If an agent has this global ability in the final calculation, this overrides any item abilities at any level. As a specific unusual example, if an agent has the global `do_anything` ability from an EveryonePermission, then giving him any item ability with is_allowed=False will have no effect.)
- `view_anything` (Similar to `do_anything`, except it expands only to abilities that start with `'view '`.)
- `edit_anything` (Similar to `do_anything`, except it expands only to abilities that start with `'edit '`.)

Below is a list of item types and the item abilities they introduce:

- Item
  - `do_anything` (Agents this ability with respect to an item automatically have every item ability for that item.)
  - `view_anything` (Similar to `do_anything`, except it expands only to abilities that start with `'view'`.)
  - `edit_anything` (Similar to `do_anything`, except it expands only to abilities that start with `'edit'`.)
  - `comment_on` (With this ability you can create comments *directly* on the item. There is no way to restrict agents from leaving *indirect* comments on an item, apart from ensuring that they don't have the ability to comment on any of the item's existing comments.)
  - `delete` (With this ability you can deactivate, reativate, or destroy the item.)
  - `view Item.name`
  - `view Item.description`
  - `view Item.creator`
  - `view Item.created_at`
  - `edit Item.name`
  - `edit Item.description`
- Agent
  - `add_contact_method` (With this ability you can create ContactMethods belonging to this Agent.)
  - `add_authentication_method` (With this ability you can create AuthenticationMethods belonging to this Agent.)
  - `login_as` (With this ability you can authenticate as this Agent.)
  - `view Agent.last_online_at`
- GroupAgent
  - `view GroupAgent.group`
- AuthenticationMethod
  - `view AuthenticationMethod.agent`
- DemeAccount
  - `view DemeAccount.username`
  - `view DemeAccount.password`
  - `view DemeAccount.password_question`
  - `view DemeAccount.password_answer`
  - `edit DemeAccount.username`
  - `edit DemeAccount.password`
  - `edit DemeAccount.password_question`
  - `edit DemeAccount.password_answer`
- Person
  - `view Person.first_name`

    - view Person.middle_names
    - view Person.last_name
    - view Person.suffix
    - edit Person.first_name
    - edit Person.middle_names
    - edit Person.last_name
    - edit Person.suffix
- ContactMethod
    - add_subscription (With this ability you can create Subscriptions belonging to this ContactMethod.)
    - view ContactMethod.agent
- EmailContactMethod
    - view EmailContactMethod.email
    - edit EmailContactMethod.email
- PhoneContactMethod
    - view PhoneContactMethod.phone
    - edit PhoneContactMethod.phone
- FaxContactMethod
    - view FaxContactMethod.fax
    - edit FaxContactMethod.fax
- WebsiteContactMethod
    - view WebsiteContactMethod.url
    - edit WebsiteContactMethod.url
- AIMContactMethod
    - view AIMContactMethod.screen_name
    - edit AIMContactMethod.screen_name
- AddressContactMethod
    - view AddressContactMethod.street1
    - view AddressContactMethod.street2
    - view AddressContactMethod.city
    - view AddressContactMethod.state
    - view AddressContactMethod.country
    - view AddressContactMethod.zip
    - edit AddressContactMethod.street1
    - edit AddressContactMethod.street2
    - edit AddressContactMethod.city
    - edit AddressContactMethod.state

  - edit AddressContactMethod.country
  - edit AddressContactMethod.zip
- Subscription
  - view Subscription.contact_method
  - view Subscription.item
  - view Subscription.deep
  - view Subscription.notify_text
  - view Subscription.notify_edit
  - edit Subscription.deep
  - edit Subscription.notify_text
  - edit Subscription.notify_edit
- Collection
  - modify_membership (With this ability you can add and remove Memberships pointing to this Collection.)
  - add_self (With this ability, you can add yourself as a member of this Collection.)
  - remove_self (With this ability, you can remove yourself as a member of this Collection.)
- Folio
  - view Folio.group
- Membership
  - view Membership.item
  - view Membership.collection
- TextDocument
  - view TextDocument.body
  - edit TextDocument.body
  - add_transclusion (With this ability, you can add a transclusion with this TextDocument as the from_item.)
- DjangoTemplateDocument
  - view DjangoTemplateDocument.layout
  - view DjangoTemplateDocument.override_default_layout
  - edit DjangoTemplateDocument.layout
  - edit DjangoTemplateDocument.override_default_layout
- FileDocument
  - view FileDocument.datafile
  - edit FileDocument.datafile
- Transclusion
  - view Transclusion.from_item
  - view Transclusion.from_item_version_number

  - view Transclusion.from_item_index
  - view Transclusion.to_item
  - edit Transclusion.from_item_index
- Comment
  - view Comment.item
  - view Comment.item_version_number
  - view Comment.from_contact_method
- TextDocumentExcerpt
  - view TextDocumentExcerpt.text_document
  - view TextDocumentExcerpt.text_document_version_number
  - view TextDocumentExcerpt.start_index
  - view TextDocumentExcerpt.length
  - edit TextDocumentExcerpt.text_document_version_number
  - edit TextDocumentExcerpt.start_index
  - edit TextDocumentExcerpt.length
- ViewerRequest
  - add_sub_path (With this ability you can create ViewerRequests with this ViewerRequest as the parent_url.)
  - view ViewerRequest.aliased_item
  - view ViewerRequest.viewer
  - view ViewerRequest.action
  - view ViewerRequest.query_string
  - view ViewerRequest.format
  - edit ViewerRequest.aliased_item
  - edit ViewerRequest.viewer
  - edit ViewerRequest.action
  - edit ViewerRequest.query_string
  - edit ViewerRequest.format
- Site
  - view Site.hostname
  - edit Site.hostname
  - view Site.default_layout
  - edit Site.default_layout
- CustomUrl
  - view CustomUrl.parent_url
  - view CustomUrl.path

- DemeSetting
  - `view DemeSetting.key`
  - `view DemeSetting.value`
  - `edit DemeSetting.value`

In order to implement permissions, Deme takes the currently authenticated Agent (anonymous or not), and decides whether it has the required ability to complete the requested action (or display some part of the view). Abilities are not just checked before doing actions, but they can also be used to filter out items on database lookups. For example, if my viewer is supposed to display a list of items I am allowed to see (because I have the `view name` ability), it will need to use permissions to filter out inappropriate results.

To modify a [X]ToOne permission, one must have the `do_anything` ability with respect to the target item. Similarly, to modify a [X]ToSome permission, one must have the `do_anything` ability with respect to the target collection. Finally, to modify a [X]ToAll permission, one must have the global `do_anything` permission.

However, there is a loophole in the setup described above. A user could simply create a collection, add a private item to that collection (because they have `do_anything` with respect to that collection), create a [X]ToSome permission for that collection (because they have `do_anything` with respect to that collection), and thus gain full access to the private item. In order to resolve this, we use the `permission_enabled` field in Membership. [X]ToSome permissions only propagate to members of the collection through memberships with `permission_enabled=True`, and agents can only modify the `permission_enabled` field of an membership if they have the `do_anything` ability with respect to the member item. By enforcing this, we guarantee that when a user modifies a [X]ToSome permission, it only affects items in the collection that were added to that collection with `permission_enabled=True` by a user that has power over that item. Since [X]ToSome permissions recursively traverse Memberships, we have a `permission_enabled` field in RecursiveMembership, that is set to true if and only if there exists a path of memberships from the parent collection to the child item all with `permission_enabled=True`.

4.2 Front-end (viewers)

4.2.1 Overview

A viewer is a Python class that processes browser or API requests. Any URL that starts with `/viewing/` is routed to a viewer (vanity URLs are also routed to viewers via ViewerRequests, but `/static/` URLs and invalid URLs are not). Each viewer defines the item type it can accept, and multiple viewers can accept the same item type (you could have ItemViewer and SuperItemViewer which both handle items). There should be a default viewer for every item type with the same name as the item type (in lowercase), and if there is none, then the default viewer of the superclass should be used. Viewers that handle item type X always handle items that are in subclasses of X.

4.2.2 URLs

Our URLs are restful. Every URL defines a viewer, an action, a noun (or none for actions on the entire item type), a format, an optional parameters in the query string. Here are some example URLs:

- /viewing/item (item viewer, default "list" action, default "html" format)
- /viewing/person/new.xml (person viewer, new action, xml format)
- /viewing/person/1 (person viewer, default "show" action, person with id=1 is the noun, default "html" format)
- /viewing/person/1/edit.json?version=5 (same as above, but json format, edit action, and version 5)

4.2.3 Actions

Every viewer defines a set of actions it responds to. Actions are divided into two groups: those that take nouns (which are always item ids) called item actions, and those that do not take nouns called item type actions. In order to make URLs unambiguous, item ids must be numbers, and action names can only be letters (although we may later decide to allow other characters, such as underscores and dashes, or even numbers that do not appear at the beginning).

An action corresponds to a single Python function. If you visit /viewing/item/list, Deme will call the type_list method of the ItemViewer class. If you visit /viewing/person/5/show, Deme will call the item_show method of the PersonViewer class. Actions return the HTTP response to go back to the browser. Actions can call other actions from other viewers to embed views in other views (for example, the DocumentViewer could embed a view from the PersonViewer to show a little profile of the author at the top).

4.2.4 Nouns

Item actions take in a noun in the URL, which is the unique id of the item it acts upon. If viewers need more information (say I submitted a form that specified multiple people I wanted to add to a group), the data is passed in the query string or the HTTP post data, and the data required is up to the specific viewer. The only query string parameters that are reserved right now by convention are "version" (which specifies a specific version of the item the viewer is acting on) and "redirect" (which specifies the URL to return to after submitting the form on this page).

4.2.5 Formats

An additional parameter is passed in defining the response format, like HTML or XML. The default is HTML. Each action specifies a different behavior for each format it accepts. For example, in the "show" action, the "html" format will display a page showing everything about the item, while the "rss" format will render an RSS document with the latest action notices. Note that the format only specifies the response format. The request format (what the browser sends to the server) is always the same: all parameters encoded in the URL or the HTTP post data. We will only be using HTTP as the transport for viewers (although we can define things that accept emails and SSH and other protocols, they just won't be called viewers).

4.2.6 Authentication

Whenever a visitor (or another web service or bot) is at an action of a viewer, he has an authenticated AuthenticationMethod, and through that AuthenticationMethod, is an Agent. If a visitor has not authenticated, they'll be using AnonymousAgent. We will support various ways of authenticating via the different subclasses of AuthenticationMethod.

4.2.7 DjangoTemplateDocuments

There is a DjangoTemplateDocument viewer right now, which accepts DjangoTemplateDocuments, and when viewed with the "render" action, it renders the DjangoTemplateDocument as HTML (or whatever format) straight back to the browser. This allows users to add web content that is not really tied to a viewer, so they can fully customize the user experience. By using DjangoTemplateDocuments and vanity URLs, a webmaster can use Deme to create a completely customized site that has no sign of Deme (unless a visitor specifically types in a /viewing/ or /static/ URL).

However, DjangoTemplateDocuments only allow the content to be customized, and not the things that a view does. For example, one cannot write a DjangoTemplateDocument to create a new record in the database, or to send out an email when visited, or more importantly, to do unauthorized things like execute UNIX commands.

Also, every HTML response from a viewer is rendered by inheriting from the default layout from the given site, so by modifying DjangoTemplateDocuments, one can change the look and feel of ordinary viewers to some extent.

4.3 Modules

Modules are self-contained collections of item types and viewers (and arbitrary Django code) that can be imported into any Deme project. They work just like Django apps, except by virtue of being in the `modules/` directory they are registered into the Deme viewer framework. All of the item types discussed in this document are part of the Deme "core" (the `cms/` directory). Modules cannot generally override or change functionality of existing parts of code (so you cannot add a button to a page rendered by ItemViewer). They can only add new functionality.

4.4 Email integration

As described in the section on Subscriptions, Deme will email notifications for every action notice made on items that are subscribed to (in the future we will support other ContactMethods, like sending SMS notifications). The communication also goes the other way: if someone responds to a notification email (or sends an email to the address corresponding to a particular item), that will become a comment on Deme.

4.5 Bootstrap Framework

Deme uses the Bootstrap 3 front-end framework (http://getbootstrap.com).

Stylesheets (CSS) for Deme are generated from LESS files (http://lesscss.org). There are many ways to compile LESS files but the one used was CodeKit (http://incident57.com/codekit), a commercial software package that simplifies the process.

In order to allow easy upgrade of the Bootstrap framework files, files in /static/less/bootstrap should not be customized for Deme as any customization would be overwritten when upgrading Bootstrap. Instead, changes should be made to files in /static/less/deme.

The Bootstrap Javascript is simply included as a minified JS file.

CHAPTER

FIVE

CODE DOCUMENTATION

This documentation file is automatically generated and kind of ugly. It will be more organized later.

5.1 Item Types (cms.models)

5.2 Item Viewers (cms.views)

CHAPTER

SIX

INDICES AND TABLES

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