Introduction to FreeNAS development

John Hixson
john@ixsystems.com
iXsystems, Inc.

Abstract

FreeNAS has been around for several years now but development on it has been by very few people. Even with corporate sponsorship and a team of full time developers, outside interest has been minimal. Not a week goes by when a bug report or feature request is not filed. Documentation on how to develop on FreeNAS simply does not exist. Currently, the only way to come up to speed on FreeNAS development is to obtain the source code, read through it, modify it and verify it works. The goal of this paper is to create a simple FreeNAS application to demonstrate some of the common methods used when dealing with FreeNAS development, as well as showcase some of the API.

1 Where to start

When learning to program on a new platform, where does one even begin? In the case of FreeNAS, the answer to this depends on what your needs are. If you want to modify a certain release, you can just grab an ISO image for said release and install it. If you would like to hack on new features, you will need to build your own image to work from. In either case you can checkout the source code for a release or the current development branch and build from it.

2 Where to get the code

FreeNAS source code can be obtained from github: http://github.com/freenas/freenas.git

3 How to build

The FreeNAS source code is built using the make(1) command. There are several targets available, however, most of the time the default “all” target is what you'll need. So let's look at a typical build:

```
# git clone http://github.com/freenas/freenas.git freenas
# cd freenas
# make
No git repo choice is set.  Please use "make git-external" to build as an
external developer or "make git-internal" to build as an iXsystems
internal developer.  You only need to do this once.
*** [git-verify] Error code 1

Stop in /usr/home/john/freenas.
```

At this point, the build will bomb out and tell you to either “make git-internal” or “make git-external”. A “git-external” build is what you will need, so:

```
# make git-external
# make
```

The build will now do a checkout of TrueOS source and a ports tree. What is TrueOS? TrueOS is FreeBSD with some of our local modifications in it. A frozen ports tree is also used so versions of the ports we use don't get bumped without us knowing. Once the checkout is complete, the build kicks off. Here is what happens:

1. buildworld
2. buildkernel
3. installworld
4. installkernel

5. debug kernel

Once the build has reached this point, a world chroot has been created and populated with FreeBSD. Now, ports need to be built. There are roughly 200 ports that get built. As each port is built, a package is also created and cached so that packages can be installed instead of building from source every build. After the ports are build, various customization functions run and then a raw disk image is created. When the disk image is completed, an ISO image is created as well as an GUI upgrade image. You are now ready to install FreeNAS!

4 Installing FreeNAS

Installing FreeNAS is easy. When developing for FreeNAS it is most convenient to install it in a virtual machine. FreeNAS will run fine in VMWare, VirtualBox, Parallels and even Bhyve. When you boot a FreeNAS ISO, the very first prompt is to install or upgrade. For this paper, we are installing. Upgrading is certainly an option and generally part of the development process though. Here is a rundown of a FreeNAS install:

1. Install
2. Pick the disk to install on
3. Proceed with installation
4. Eject ISO
5. Shutdown

Now, you are ready to boot FreeNAS!

Configuring FreeNAS:

When FreeNAS boots up, you will be dropped into a menu with several options. At the very minimum, a network interface must be configured along with DNS. Choose option 1 to configure the network interface and follow the instructions. Afterwards, choose option 6 to configure DNS and follow the instructions. Now you can reach FreeNAS in your web browser.

5 How to create a module

FreeNAS is primarily written in python using the django framework. So to make any kind of interface changes, you will be required to have some understanding of django. The UI for FreeNAS lives under /usr/local/www/freenasUI. To do any kind of work under this directory means you will have to mount the root file system read/write. So, let's create a module!

```
# mount -uw /
# cd /usr/local/www/freenasUI
# python manage.py startapp bsdcan
# ls bsdcan__init__.py
models.py
tests.py
views.py
```

You will see the following files:

- __init__.py
  - Not heavily used in FreeNAS but can be if using django apps as modules
- models.py
  - Represents data in the database
- tests.py
  - Not heavily used in FreeNAS, but can be used for unit tests
- views.py
  - Represents the “view” in the UI

These files are created when you run the manage.py command. FreeNAS has a few additional files that will need to be created for it to work with the UI:

- forms.py
  - “forms” that represent the model, often created in the view functions
- urls.py
  - List of URL's that get matched and what view to call
- nav.py
  - Placement of items in the navtree menu
- admin.py
  - How to display data in a datagrid (if using a datagrid).

Now that the files have been explained, we're going to create a very simple FreeNAS application. The application will display 'BSDCan' in the navtree. When you click on it, it will have an 'BSDCan 2014' subtree, under which messages can be created and displayed. This application is very silly, but will demonstrate how you create a FreeNAS application, display it in the navtree with various options, and how the UI interfaces with the applications.

```
# touch forms.py urls.py nav.py admin.py
```

First, a model must be created so that it can be represented in the database, so the models.py file is edited. In this example, the “BSDCan” class is created. It inherits from the freeadmin “Model” class which is the django “Model” class with some additional methods required by FreeNAS. This model will represent how data for this application is stored in the sqlite database used by FreeNAS. In this example, only a message with a length of 1024 bytes will be stored. Here are the meanings to the fields in the class:

a_msg – Name of the property, it is of the type models.CharField()
max_length – Size of the field
verbose_name – What gets displayed in front of the field on a form
help_text – What gets displayed when the help icon is hovered over

The FreeAdmin class is used to set different meta-data used by the freeadmin app that handles must of the FreeNAS interface. In this case, icons are being set to be displayed in the navtree for the model.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _
from freenasUI.freeadmin.models import Model

class BSDCan(Model):
    a_msg = models.CharField(
        max_length=1024,
        verbose_name=_("Message"),
        help_text=_("BSDCan message to display")
    )

    class Meta:
        verbose_name = _("BSDCan")
        verbose_name_plural = _("BSDCan")

    class FreeAdmin:
        icon_model = 'BobbleIcon'
        icon_object = 'BobbleIcon'
        icon_view = 'BobbleIcon'
        icon_add = 'BobbleIcon'
```

Once a FreeNAS model has been defined, it can be added to the database as a table. There are two steps to this process. The first is a schemamigration. If it is the first time doing a schemamigration for the application, django must be told so using the “--initial” flag. Otherwise, “--auto” can be used which will pick up any additional fields that may have been added to the model.

```
# python manage.py schemamigration bsdcan –initial
 + Added model bsdcan.BSDCan
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate bsdcan
```

What this does is verify that everything is okay and no default values need to be set. Once that is verified, a "migration" script is generated in migrations/0001_initial.py:

```
# -*- coding: utf-8 -*-
import datetime
from south.db import db
from south.v2 import SchemaMigration
from django.db import models


class Migration(SchemaMigration):

    def forwards(self, orm):
        # Adding model 'BSDCan'
        db.create_table(u'bsdcan_bsdcan', (
            (u'id', self.gf('django.db.models.fields.AutoField')(primary_key=True)),
            ('a_msg', self.gf('django.db.models.fields.CharField')(max_length=1024)),
        ))
        db.send_create_signal(u'bsdcan', ['BSDCan'])


    def backwards(self, orm):
        # Deleting model 'BSDCan'
        db.delete_table(u'bsdcan_bsdcan')


    models = {
        u'bsdcan.bsdcan': {
            'Meta': {'object_name': 'BSDCan'},
            'a_msg': ('django.db.models.fields.CharField', [], {'max_length': '1024'}),
            u'id': ('django.db.models.fields.AutoField', [], {'primary_key': 'True'})
        }
    }

    complete_apps = ['bsdcan']
```

The migration script provides to migrate to the new schema, and to migrate backwards if necessary as well. To perform the actual migration, the following command must be run:

```
# python manage.py migrate bsdcan
Running migrations for bsdcan:
 - Migrating forwards to 0001_initial.
 > bsdcan:0001_initial
 - Loading initial data for bsdcan.
Installed 0 object(s) from 0 fixture(s)
```

Now this model is represented as a table in the sqlite database. To confirm this, look at the database:

```
# sqlite3 /data/freenas-v1.db
SQLite version 3.8.0.2 2013-09-03 17:11:13
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables bsdcan_bsdcan
bsdcan_bsdcan
```

```
sqlite> .schema bsdcan_bsdcan
CREATE TABLE "bsdcan_bsdcan" ("id" integer NOT NULL PRIMARY KEY, "a_msg" varchar(1024) NOT NULL);
```

Almost all models in FreeNAS need a form model as well. Most of the time, the form will have more fields then there are in the model along with additional method overrides to clean the passed in data and save it. In this example, we keep things simple and just tell FreeNAS to make the form the same as the model.

```
from freenasUI.common.forms import ModelForm
from freenasUI.bsdcan import models

class BSDCanForm(ModelForm):
    class Meta:
        model = models.BSDCan
```

In order for FreeNAS to know where our application is located, we have to tell django where it resides. This is done in the urls.py file. The majority of URL patterns are just a regular expression, a name for it, and the name of the view that is called when the regular expression is matched. This example matches the FreeNAS URL + /home/", names the URL "bsdcan_home", and specifies the view function "bsdcan_home" get called. Naming of a URL pattern is used in django so that "bsdcan_home" can be used instead of a URL path.

```
from django.conf.urls import patterns, url

urlpatterns = patterns('freenasUI.bsdcan.views',
    url(r'^home/$', 'bsdcan_home', name="bsdcan_home"),
)
```

Here is the "bsdcan_home" view that was specified in urls.py. All views are in views.py. All view functions take a request object as a parameter. Additional parameters can be specified in the urls.py file. Often times, the unique ID for the object being referenced is matched in urls.py and a parameter to the view function.

This is a typical view for a form that handles GET and POST methods. If the request method is GET, an empty form is created and rendered. If the request method is POST, the form is checked to be valid, saved, and a message is returned indicating the operation was successful. The render() call relies on an HTML template directory for bsdcan to exist and that the home.html file exists.

```
from django.shortcuts import render
from django.utils.translation import ugettext_lazy as _
from freenasUI.freeadmin.views import JsonResp
from freenasUI.bsdcan import forms

def bsdcan_home(request):

    if request.method == 'POST':
        form = forms.BSDCanForm(request.POST)
        if form.is_valid():
            form.save()
            return JsonResp(
                request,
                message=_("BSDCan successfully added.")
            )
    else:
        form = forms.BSDCanForm()

    return render(request, "bsdcan/home.html", {
        'form': form
    })
```

Here is the contents of templates/bsdcan/home.html. This just "inherits" the freeadmin/generic_form.html file.

Since nothing is being added on here, it's simple. However, there is much flexibility with templates just as with classes. Templates can be inherited and properties within them can be overridden.

```
{% extends "freeadmin/generic_form.html" %}
```

The last part of this example is the navtree code. The job of the navtree is to display menus in a tree structure. Generally there will be a top level item with an icon with items or sub menus beneath it that contain operations to add, edit, configure and delete objects. Some items will just be a form that pops up, others a datagrid, and still others a tab in a multi pane window. All of the code goes into nav.py, which is a dynamically loaded TreeNode() object. So at the top level, a NAME and ICON will be defined. There are other properties available, but this is a simple example. For further reference, the freeadmin/tree/tree.py and freeadmin/navtree.py files can be checked out. What this does is show "BSDCan" with a beastie icon in the navtree. To show menus beneath it, more classes will need to be defined. So the "BSDCanView" class is defined. It sets it's name to "BSDCan 2014", which is what will show beneath the "BSDCan" menu. In it's init method, it creates a node where new messages can be added. This is displayed beneath "BSDCan 2014". The init method then grabs all the messages (if any) that exist in the database and creates nodes for them and appends it beneath the "BSDCan 2014" menu. The additional fields in the for loop are as follows:

gname – Unique identifier for this TreeNode
name – Text displayed in the navtree
type – Tells the Javascript layer what function to call
view – The view to use
kwargs – Arguments to pass to the view
app_name – The name of the application

The magic that is occurring here is in the type, view and kwargs properties. The type is saying to use the Javascript editObject() function. The view 'freeadmin_bsdcan_bsdcan_edit' is a special view created for all models when FreeNAS starts. The syntax is "freeadmin_APPLICATION_MODEL_(add|edit|delete|datagrid)". Whenever calling an edit or delete function, the unique ID needs to be passed as an argument so it's set in the kwargs field.

```
from django.utils.translation import ugettext_lazy as _
from freenasUI.bsdcan import models
from freenasUI.freeadmin.tree import TreeNode

NAME = _('BSDCan')
ICON = 'BeastieIcon'

class BSDCanView(TreeNode):
   gname = 'BSDCan'
   name = _(u'BSDCan 2014')
   icon = 'BobbleIcon'

   def __init__(self, *args, **kwargs):
      super(BSDCanView, self).__init__(*args, **kwargs)

      node = TreeNode()
      node.name = _(u'Add message')
      node.type = 'object'
      node.view = 'freeadmin_bsdcan_bsdcan_add'
      node.icon = 'SettingsIcon'
      self.append_child(node)

      msgs = models.BSDCan.objects.all()
      for m in msgs:
         node = TreeNode()
         node.gname = m.a_msg
         node.name = m.a_msg
         node.type = 'editobject'
         node.view = 'freeadmin_bsdcan_bsdcan_edit'
         node.kwargs = {'oid': m.id}
         node.model = 'BSDCan'
         node.icon = 'BobbleIcon'
```

```
        node.app_name = 'BSDCan'
        self.append_child(node)
```

Now that the application is complete, restart django so that the changes are visible in the UI.

```
# service django restart
```

6 Testing changes

Testing changes is no different than any other kind of web development. The only difference on FreeNAS is that if any changes are made to python code, django will need to be restarted. Changes made to HTML templates or Javascript do not require a restart. Javascript changes require a refresh whereas HTML template changes require no restart or refresh.

7 Debugging

Most debugging on FreeNAS comes from the python logging module. Just about every file has an import logging statement with it's on unique name. Logging for django is configured in django's settings.py file. DEBUG can also be set to TRUE so that when a crash occurs you get a python stack trace that helps isolate the issue. Other useful debugging methods are chrome and Firefox build in Javascript debuggers for when Ajax methods are crashing but it isn't immediately obvious from the UI. Lots of useful information is also written out to the following logs:

/var/log/messages
/var/log/nginx-access.log
/var/log/nginx-error.log

Usually, if django won't even start, a python stack trace indicating why can be found in the nginx error log. /var/log/debug.log can also be used but must be uncommented in /etc/syslog.conf and syslogd restarted.

How to get involved

FreeNAS is always looking for help! If you are interested in joining the project, helping out or submitting patches, go to http://www.freenas.org and go to 'Our community'. Source code can be obtained from http://github.com/freenas/freenas.git. If you have patches, please send a pull request.

8 Conclusion

FreeNAS is a very powerful operating system. It has become very popular. With popularity comes many more people using it and that means more bug reports and feature requests. The people who bring you FreeNAS are a small team and need help. It is my hope that by demonstrating this simple application that more developers will be interested in hacking on FreeNAS.