RapidSMS Documentation

Release 0.21.0

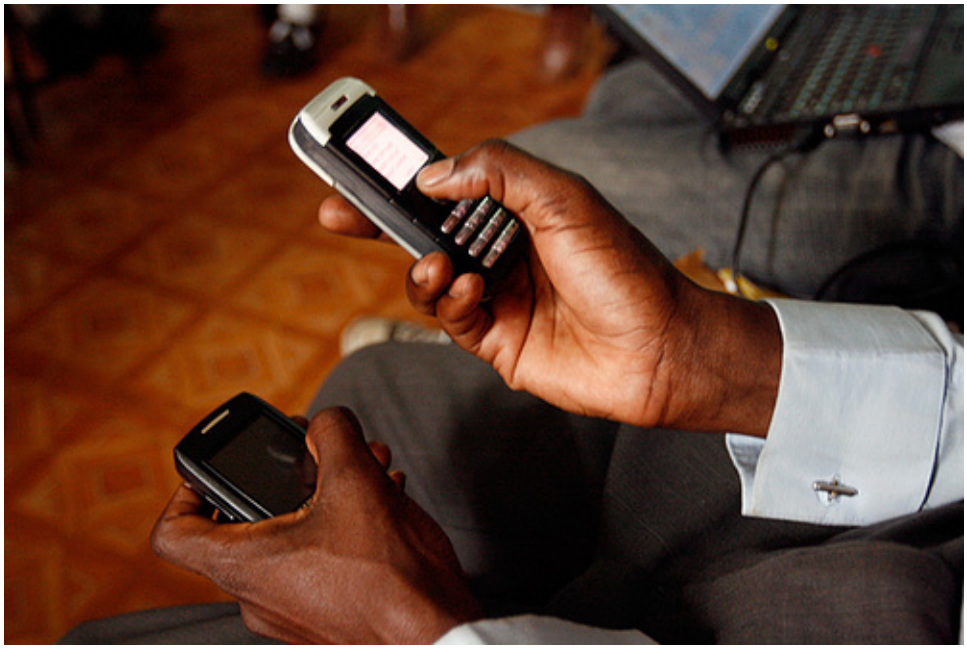
RapidSMS

July 24, 2015

Contents

CHAPTER 1

RapidSMS Overview



RapidSMS is a free and open-source framework for dynamic data collection, logistics coordination and communication, leveraging basic short message service (SMS) mobile phone technology. It can be used by anyone and because one size does not fit all and no two projects are exactly the same, RapidSMS is easily customized to meet the specific needs of the project and is scalable at an enterprise level. It is currently being utilized by large multilateral organizations (such as the United Nations), development professionals (such as the Earth Institute at Columbia University), and small NGOs and CBOs (such as Tostan).

RapidSMS is written in Python and Django and is a *framework* for building highly customized applications. While there are increasingly more and more pre-configured applications being created for RapidSMS, most projects will continue to benefit from applications designed specifically to meet the need and demands of their stakeholders.

1.1 RapidSMS at a glance

The goal of this section is to give you enough technical specifics to understand how RapidSMS works, but this isn't intended to be a tutorial or reference. When you're ready to start a project, you can install RapidSMS and begin writing your own custom applications.

As a quick example, here's how we might create a simple application, written in Python, that replies 'pong' after receiving the message 'ping':

```
from rapidsms.apps.base import AppBase

class PingPong(AppBase):

    def handle(self, msg):
        if msg.text == 'ping':
            msg.respond('pong')
            return True
        return False
```

1.2 This is just the surface

This has been only a quick overview of RapidSMS's functionality. The next obvious steps are for you to install RapidSMS, read the tutorial, and *join the community*. Thanks for your interest!

CHAPTER 2

Installing RapidSMS

Note: RapidSMS depends on the Django web framework. If you're new to Django, we recommend reading through the Django installation instructions before installing RapidSMS.

The recommended way to install RapidSMS is with Pip (since RapidSMS is available on PyPI):

```
pip install rapidsms
```

2.1 Starting a New RapidSMS Project

2.1.1 Installing the RapidSMS project template

If you're starting a new RapidSMS project, you can use the RapidSMS project template. The template is a custom project template for Django 1.8 and therefore requires Django 1.8 or later to use.

To use the project template, first make sure you have the latest version of Django installed:

```
pip install "Django>=1.8,<1.9"
```

Now you can use the `startproject` management command with the `template` option. You just need to specify your project name at the end of the command:

```
django-admin.py startproject --template=https://github.com/rapidsms/rapidsms-project-template/zipball
```

This will create a new project using the name you specified. Inside your project, you'll find a *README.rst* file with instructions to setup your project.

2.1.2 Installing the latest development version

The latest development version is available in our Git repository. Get it using this shell command, which requires Git:

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

You can also download a zipped archive of the development version.

CHAPTER 3

RapidSMS Tutorial

This tutorial will walk you through the creation of a simple RapidSMS application.

RapidSMS builds on the Django web development frameowork. We assume that you are familiar with Django, at least the material in the Django tutorial. If you haven't worked through that before, please start there, then come back here when you feel comfortable with the concepts there.

3.1 Outline

- *RapidSMS Tutorial Part 1*: Start a new RapidSMS project. Set up message tester. Write a minimal app that responds to a message. Put it through its paces.
- *RapidSMS Tutorial Part 2*: Configure the default app with a custom response. Demonstrate keyword and pattern handlers.
- *RapidSMS Tutorial Part 3*: Make a RapidSMS app that uses Django to store and update data.
- *RapidSMS Tutorial Part 4*: It's probably time to actually send and receive messages to telephones. Tropo has free development accounts and there's a Tropo backend at https://github.com/caktus/rapidsms-tropo. Walk through adding that to the project. Set up a Tropo development account. Demo the test app from step 2, this time using real messages.

Start with *RapidSMS Tutorial Part 1*.

3.1.1 RapidSMS Tutorial Part 1

In this part of the tutorial, we will:

- start a new RapidSMS project
- set up message tester
- write a minimal application that responds to a message
- demonstrate our application

Start a project

We're going to create a new Django project, using the RapidSMS project template at https://github.com/rapidsms/rapidsms-project-template.

Install Django

But before we can do that, we need to have Django installed, so we can use the Django startproject command. So we'll start by creating the virtualenv we'll use, activating it, and installing Django into it:

```
~ $ virtualenv rapidsms-tut-venv
Running virtualenv with interpreter /usr/bin/python2.7
New python executable in rapidsms-tut-venv/bin/python2.7
Also creating executable in rapidsms-tut-venv/bin/python
Installing distribute.....................................................................................
Installing pip...............done.
~ $ . rapidsms-tut-venv/bin/activate
(rapidsms-tut-venv)~ $ pip install Django
Downloading/unpacking Django
[...]
Successfully installed Django
Cleaning up...
(rapidsms-tut-venv)~ $
```

Start the project

Now we'll use the Django startproject command, with the RapidSMS project template:

```
(rapidsms-tut-venv)~ $ django-admin.py startproject --template=https://github.com/rapidsms/rapidsms-p
(rapidsms-tut-venv)~ $ cd rapidsms_tut
(rapidsms-tut-venv)~/rapidsms_tut $ tree
.
-- manage.py
-- rapidsms_tut
|   -- __init__.py
|   -- settings.py
|   -- templates
|   |   -- rapidsms
|   |       -- _nav_bar.html
|   -- urls.py
|   -- wsgi.py
-- README.rst
-- requirements
    -- base.txt

4 directories, 8 files
(rapidsms-tut-venv)~/rapidsms_tut $
```

Install dependencies

Install the dependencies:

```
(rapidsms-tut-venv)~/rapidsms_tut $ pip install -r requirements/base.txt
[... lots of output omitted ...]
Successfully installed RapidSMS South requests django-tables2 djappsettings django-selectable
Cleaning up...
(rapidsms-tut-venv)~/rapidsms_tut $
```

Remove some unneeded applications

The RapidSMS project template installs a number of applications by default. Let's disable some to simplify things. In `rapidsms_tut/settings.py`, comment out the following lines:

```
--- a/rapidsms_tut/settings.py
+++ b/rapidsms_tut/settings.py
@@ -202,7 +202,7 @@ INSTALLED_APPS = (
     "rapidsms.contrib.messagelog",
     "rapidsms.contrib.messaging",
     "rapidsms.contrib.registration",
-    "rapidsms.contrib.echo",
+    # "rapidsms.contrib.echo",
     "rapidsms.contrib.default",  # Must be last
 )

@@ -215,6 +215,6 @@ INSTALLED_BACKENDS = {
 LOGIN_REDIRECT_URL = '/'

 RAPIDSMS_HANDLERS = (
-    'rapidsms.contrib.echo.handlers.echo.EchoHandler',
+    # 'rapidsms.contrib.echo.handlers.echo.EchoHandler',
+    # 'rapidsms.contrib.echo.handlers.ping.PingHandler',
 )
```

Set up the database

The default settings in the RapidSMS project template use SQLite as the database. You should *never* use SQLite in production, but we'll leave it configured here for simplicity.

Initialize our database. First we use syncdb. Go ahead and create a superuser when prompted:

```
(rapidsms-tut-venv)~/rapidsms_tut $ python manage.py syncdb
Syncing...
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table south_migrationhistory

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'username'):
Email address: username@example.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

```
Synced:
 > django.contrib.auth
 > django.contrib.contenttypes
 > django.contrib.sessions
 > django.contrib.sites
 > django.contrib.messages
 > django.contrib.staticfiles
 > django.contrib.admin
 > django_tables2
 > selectable
 > south
 > rapidsms.contrib.handlers
 > rapidsms.contrib.httptester

Not synced (use migrations):
 - rapidsms
 - rapidsms.backends.database
 - rapidsms.contrib.messagelog
(use ./manage.py migrate to migrate these)
(rapidsms-tut-venv)~/rapidsms_tut $
```

Then we apply migrations using South's migrate command:

```
(rapidsms-tut-venv)~/rapidsms_tut $ python manage.py migrate
Running migrations for rapidsms:
[...]
 - Loading initial data for rapidsms.
Installed 0 object(s) from 0 fixture(s)
Running migrations for database:
[...]
 - Loading initial data for database.
Installed 0 object(s) from 0 fixture(s)
Running migrations for messagelog:
[...]
 - Loading initial data for messagelog.
Installed 0 object(s) from 0 fixture(s)
(rapidsms-tut-venv)~/rapidsms_tut $
```

Start the server

We should now be ready to start our project. It won't do much yet, but we can see if what we've done so far is working:

```
(rapidsms-tut-venv)~/rapidsms_tut $ python manage.py runserver
Validating models...

0 errors found
May 03, 2013 - 09:47:56
Django version 1.5.1, using settings 'rapidsms_tut.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

And if you go to http://127.0.0.1:8000/ with a local browser, you should see a prompt to login. Entering the userid and password you used earlier to create a superuser should work and you'll see RapidSMS's "Installation Successful!" page.

Message Tester

Let's take a look at one of the contributed applications that is installed by default, Message Tester. There should be a link to it at the top of the page, or you can just go to http://localhost:8000/httptester/.

With Message Tester, you can manually enter and send a message to your RapidSMS site as if it came from outside. Let's try it out. In the Phone Number field, change our phone number to "123456". (The phone number doesn't really matter, but it'll make your output match what we show here in the tutorial.) Then in the Single Message field, enter "ping" and click the "Send" button.

On the right side of the page, Message Tester shows the messages sent and received, in reverse order (so the most recent message is first). Here's what you might see:

```
05/03/2013 9:54 a.m.        123456« Sorry, RapidSMS could not understand your message.
05/03/2013 9:54 a.m.        123456» ping
```

The "123456»" indicates that a message was sent from phone number 123456 to RapidSMS. The text of the message was "ping".

The "123456«" tells us that RapidSMS sent a message to phone number 123456. The content of that message was "Sorry, RapidSMS could not understand your message." That shouldn't be too surprising, since we haven't written an application yet. But then, where did the "Sorry" message come from? That comes from RapidSMS's default handler, which we'll learn more about later.

(If instead of the "Sorry" message, you get a response of "pong", that just means you missed the step above of commenting out a few lines in `settings.py` that the RapidSMS project template installs by default. If you go back and make that change, restart your app, and try again, it should work.)

A minimal application

The Applications Overview shows a trivial RapidSMS application:

```
1  from rapidsms.apps.base import AppBase
2
3  class PingPong(AppBase):
4
5      def handle(self, msg):
6          if msg.text == 'ping':
7              msg.respond('pong')
8              return True
9          return False
```

Let's see how we would add that to our project.

A RapidSMS app must first be a Django app, so let's create an empty Django app. We'll call it *tut*:

```
(rapidsms-tut-venv)~/rapidsms_tut $ python manage.py startapp tut
(rapidsms-tut-venv)~/rapidsms_tut $ tree tut
tut
-- __init__.py
-- models.py
-- tests.py
-- views.py

0 directories, 4 files
(rapidsms-tut-venv)~/rapidsms_tut $
```

Now we need to add our app to Django's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
   [...]
    # RapidSMS
    "tut",
   [...]
    "rapidsms.contrib.default",  # Must be last
)
```

Our RapidSMS app class must be in a file named `app.py` in our Django application's directory, so create a file `rapidsms_tut/tut/app.py` and paste the code from above. Here's what it should look like when you're done:

```
(rapidsms-tut-venv)~/rapidsms_tut $ cat tut/app.py
from rapidsms.apps.base import AppBase

class PingPong(AppBase):

    def handle(self, msg):
        if msg.text == 'ping':
            msg.respond('pong')
            return True
        return False
(rapidsms-tut-venv)~/rapidsms_tut $
```

Try our application

Now, let's start our project again and try it out. Start Django as before, go to the Message Tester app, and send a message containing "ping" (exactly, it must be all lower-case). Instead of "RapidSMS could not understand your message", this time your app responds "pong":

```
05/03/2013 10:49 a.m.       123456« pong
05/03/2013 10:49 a.m.       123456» ping
```

You can find a brief explanation of how this app works in the Applications Overview.

Continue with *RapidSMS Tutorial Part 2*.

3.1.2 RapidSMS Tutorial Part 2

We'll continue the tutorial by introducing the RapidSMS default app. Then we'll show how using RapidSMS handlers can handle parsing incoming messages for you.

Default Application

In *part 1*, we saw the *default application* doing its work, responding to messages that no other application had handled. It's a good idea to keep the default application at the end of `INSTALLED_APPS` so that it can give some response when your application doesn't recognize a message. Otherwise your users will get no response and won't know there was a problem. Or worse, the default application will respond to the message before your app sees it, confusing the user.

You can change the response used by the default application by changing *`DEFAULT_RESPONSE`*. For example, if you've implemented a HELP command in your project, you might change the default response to:

```
DEFAULT_RESPONSE = "Sorry, %(project_name)s could not \
understand your message.  Send HELP to get a list of \
valid commands."
```

Of course, you could also just send the help in the default response.

"Handling" a Message

We said the default application would respond if no other application had handled the message, but how does RapidSMS know that an application has "handled" the message?

One way is for an application's *handle* method to return `True`. That tells RapidSMS that the application has handled the message and no other applications need to try to handle it too. On the other hand, if the application returns `False`, RapidSMS will continue passing the message to applications in its list until one returns `True` or it runs out of applications.

That's why the default application should be kept at the end of the `INSTALLED_APPS`, because we don't want RapidSMS to call the default application until it has tried every other one, and RapidSMS calls the applications in the order of `INSTALLED_APPS`.

RapidSMS Handlers

There are a few very common cases for RapidSMS, such as looking for messages starting with a particular word, or messages that match a particular pattern. Instead of writing that code over and over yourself, you can use RapidSMS handlers.

Using handlers has three steps:

1. Write one or more subclasses of handler classes.
2. Add *"rapidsms.contrib.handlers"* to `INSTALLED_APPS`.
3. Add the full classnames of each of your new classes to *`RAPIDSMS_HANDLERS`*.

By the way, RapidSMS handlers are just implemented as another app, `rapidsms.contrib.handlers`. Which shows what you can do with a RapidSMS app.

Keyword Handlers

We mentioned earlier that you might want to implement a HELP command for your users. We can do that using a *Keyword Handler*.

You'll write a class that subclasses *`KeywordHandler`*. Your keyword will be "help" (it's not case sensitive). If someone sends just "HELP", we'll respond with a message telling them how to get more help. If someone sends "HELP something", we'll give them more specific help if we can, and otherwise send the same response we would to a bare "HELP".

Create a file `myhandlers.py` with the following content:

```
# myhandlers.py

from rapidsms.contrib.handlers import KeywordHandler

help_text = {
    'aaa': 'Help for aaa',
    'bbb': 'Help for bbb',
    'ccc': 'Help for ccc',
}


class HelpHandler(KeywordHandler):
```

```
    keyword = "help"

    def help(self):
        """Invoked if someone just sends `HELP`.  We also call this
        from `handle` if we don't recognize the arguments to HELP.
        """
        self.respond("Allowed commands are AAA, BBB, and CCC. Send "
                     "HELP <command> for more help on a specific command.")

    def handle(self, text):
        """Invoked if someone sends `HELP <any text>`"""
        text = text.strip().lower()
        if text == 'aaa':
            self.respond(help_text['aaa'])
        elif text == 'bbb':
            self.respond(help_text['bbb'])
        elif text == 'ccc':
            self.respond(help_text['ccc'])
        else:
            self.help()
```

Now, add *"rapidsms.contrib.handlers"* to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.handlers",
    ...
]
```

and add your new class to *RAPIDSMS_HANDLERS*:

```
RAPIDSMS_HANDLERS = [
    ...
    "myhandlers.HelpHandler",
    ...
]
```

Now, if you start RapidSMS and send a message "HELP", you should get this response:

```
Allowed commands are AAA, BBB, and CCC. Send HELP <command> for more help on a specific command.
```

and if you send "HELP AAA", you should get whatever help is available for AAA.

Handlers Must Handle

Warning: When a handler is called for a message, the handler must handle the message itself, because no other handlers or apps will be called. Since this handler matched the message, RapidSMS expects that this handler will take care of the message. If you need more flexibility, you'll need to write a normal RapidSMS application.

Pattern Handlers

A *Pattern Handler* is like a keyword handler, but with two differences:

1. The pattern can match any part of the message, not just the beginning

2. Groups can be used in the regular expression to help parse the message. Whatever matches the groups is passed to your handler.

Note: Be careful when deciding to use a pattern handler. Your regular expression needs to be flexible enough to cope with any message someone might send that you want your handler to handle.

Here's an example from the *PatternHandler* documentation. You can send a message like "5 plus 3" and it will respond "5+3 = 8". Note that you cannot send "5 + 3" or "5plus3" or "5 plus 3 "; none of those match this simple regular expression, so this handler won't be invoked.

Add this code to your `myhandlers.py` file:

```
from rapidsms.contrib.handlers import PatternHandler

class SumHandler(PatternHandler):
    pattern = r'^(\d+) plus (\d+)$'

    def handle(self, a, b):
        a, b = int(a), int(b)
        total = a + b

        self.respond(
            "%d+%d = %d" %
            (a, b, total))

>>> SumHandler.test("1 plus 2")
['1+2 = 3']
```

and add the new class to *RAPIDSMS_HANDLERS*:

```
RAPIDSMS_HANDLERS = [
    ...
    "myhandlers.HelpHandler",
    "myhandlers.SumHandler",
    ...
]
```

Restart your app, and try sending some messages. `1 plus 2` should get a response of `1+2 = 3`. `1+2` should get the default response, because it doesn't match any of the patterns or keywords of your defined handlers, and no other RapidSMS app is going to process the message.

Continue with *RapidSMS Tutorial Part 3*.

3.1.3 RapidSMS Tutorial Part 3

So far, nothing we've done really requires Django. Let's create a RapidSMS application that uses Django's abilities to store data.

We'll create an extremely simple voting application. It will understand two messages: `VOTE <choice>` will add a vote for the specified choice, and `RESULTS` will respond with the current number of votes for each choice.

(Please notice that this application is more appropriate for a group to choose a place to go to lunch than for anything serious. It makes no attempt whatsoever to enforce any of the controls a real election would need.)

Create the application

Create a new Django application. Let's call it "voting":

```
$ python manage.py startapp voting
```

The models

This application is so simple that we'll only need one model. We'll call it `Choice`, and there will be one instance for each possible choice. Each record will contain the name of the choice and the number of votes for it so far.

```
# voting/models.py
from django.db import models

class Choice(models.Model):
    name = models.CharField(max_length=40, unique=True)
    votes = models.IntegerField(default=0)
```

Application Design

Even a simple application like this can demonstrate an important design principle for RapidSMS applications.

Instead of adding to a candidate's vote count each time a vote arrived, we could instead have created a Vote model and stored a record of each vote. That seems like a little simpler way to handle an incoming vote.

However, if we did that, whenever we needed the results we would have to query every record in our database to count up the votes for each choice. There are SQL queries that can simplify doing that, but the database still has to look at every record. And the next time we wanted the results, we'd have to do that again.

We're better off doing a little more processing on each message, if that can save us a lot of work later on.

Admin

We'll want to use the Django admin to set up our choices. Typically you would need to make a few simple changes to your project's `urls.py`, but the RapidSMS project template has already done that for us. So all we need to do is add `admin.py` to our application:

```
# voting/admin.py
from django.contrib import admin

from .models import Choice

admin.site.register(Choice)
```

The results handler

Let's start with the simpler message to handle, `RESULTS`. This is easily implemented as a RapidSMS keyword handler. Let's create a file `handlers.py` to contain our handlers, and write a handler that responds with the current votes.

```
# voting/handlers.py

from rapidsms.contrib.handlers import KeywordHandler

from .models import Choice


class ResultsHandler(KeywordHandler):
    keyword = "results"
```

```
    def help(self):
        """help() gets invoked when we get the ``results`` message
        with no arguments"""
        # Build the response message, one part per choice
        parts = []
        for choice in Choice.objects.all():
            part = "%s: %d" % (choice.name, choice.votes)
            parts.append(part)
        # Combine the parts into the response, with a semicolon after each
        msg = "; ".join(parts)
        # Respond
        self.respond(msg)

    def handle(self, text):
        """This gets called if any arguments are given along with
        ``RESULTS``, but we don't care; just call help() as if they
        passed no arguments"""
        self.help()
```

If the choices are "Moe", "Larry", and "Curly", the response to a `RESULTS` message might look like `Moe: 27; Larry: 15; Curly: 98`.

The vote handler

The `VOTE` message is slightly more work. If we receive `VOTE xxxx` where xxx is one of the choices (case-insensitive), we want to increment the votes for choice `xxx` and respond telling the user that their vote has been counted. If we receive any other message starting with `VOTE`, we'll respond with some help to tell them how the command works and what the choices are.

```
# voting/handlers.py (continued)
from django.db.models import F

class VoteHandler(KeywordHandler):
    keyword = "vote"

    def help(self):
        """Respond with the valid commands.  Example response:
        ``Valid commands: VOTE <Moe|Larry|Curly>``
        """
        choices = "|".join(Choice.objects.values_list('name', flat=True))
        self.respond("Valid commands: VOTE <%s>" % choices)

    def handle(self, text):
        text = text.strip()
        # look for a choice that matches the attempted vote
        try:
            choice = Choice.objects.get(name__iexact=text)
        except Choice.DoesNotExist:
            # Send help
            self.help()
        else:
            # Count the vote. Use update to do it in a single query
            # to avoid race conditions.
            Choice.objects.filter(name__iexact=text).update(votes=F('votes')+1)
            self.respond("Your vote for %s has been counted" % text)
```

Settings

We need to add our Django app to `INSTALLED_APPS` and our handlers to *`RAPIDSMS_HANDLERS`*:

```
INSTALLED_APPS = (
    [...]
     # RapidSMS
     "voting",
    [...]
     "rapidsms.contrib.default",  # Must be last
)

RAPIDSMS_HANDLERS = [
     [...]
     "voting.handlers.ResultsHandler",
     "voting.handlers.VoteHandler",
     [...]
]
```

Update database

We've added a new model, so we need to update our database to include it:

```
$ python manage.py syncdb
Syncing...
Creating tables ...
Creating table voting_choice
[... rest of output omitted ...]
```

Create some choices

Now it's time to start our application and create some choices to vote for.

```
$ python manage.py runserver
Validating models...

0 errors found
May 07, 2013 - 08:28:44
Django version 1.5.1, using settings 'rapidsms_tut.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Go to http://127.0.0.1:8000/admin/voting/choice/, login as the superuser you created in part 1 of the tutorial, and you should be able to add some choices.

Vote

Let's start by checking that there are no votes. Go to the message tester application (http://127.0.0.1:8000/httptester/) and send the message `RESULTS`. You should see a response showing no votes, something like this:

```
05/07/2013 8:30 a.m.        349911« Moe: 0; Larry: 0; Curly: 0
05/07/2013 8:30 a.m.        349911» RESULTS
```

(Recall that the messages are shown in reverse order.)

Now let's cast a vote. Send `VOTE Moe` and you should see something like:

```
05/07/2013 8:32 a.m.        349911« Your vote for Moe has been counted
05/07/2013 8:32 a.m.        349911» VOTE Moe
```

and if you check the results again:

```
05/07/2013 8:33 a.m.        349911« Moe: 1; Larry: 0; Curly: 0
05/07/2013 8:33 a.m.        349911» RESULTS
```

Continue with *RapidSMS Tutorial Part 4*.

3.1.4 RapidSMS Tutorial Part 4

In this part of the tutorial, we'll show one way to move beyond the message tester to send and receive text messages with real phones.

We won't be creating any new RapidSMS apps in this part. Instead, we'll update our settings so that our existing apps can send and receive actual text messages.

Online Providers

A common way to connect your RapidSMS application to the telephone system is to use an online service. Typically such a service will provide an HTTP interface that lets you send messages, and a phone number that can receive messages. When a message is received, the service will deliver it to your application via HTTP request as well.

There are other options, such as physical devices that connect to your computer. You can get an idea of some of the options by looking at the *RapidSMS Backends* documentation.

Network considerations

For an online provider to deliver messages to your site, the provider has to be able to connect to your site. If your site is not going to be accessible on the public Internet, you'll have to find an alternative way to send and receive messages.

Even if your site will be on the Internet, you might be doing your development behind a firewall, where your provider cannot connect to your development system. This can make testing your site difficult.

If you can't use a test system that's accessible on the Internet, and you want to receive messages on your development system, you'll have to get an externally visible port forwarded to the port on your development system that your site is running on.

With some providers, you can at least send outgoing messages from behind a firewall without having to arrange for incoming connections to work. Unfortunately, the one we're going to use as an example isn't one of them. It has other advantages though.

Tropo

For this example we'll use Tropo. There's a Tropo RapidSMS backend we can use, and if you're in the United States, you can get a free developer account that includes a phone number and enough free messages to try out the service. Tropo also has service in the rest of North America and western Europe, though the free developer account is not available. If you're outside Tropo's service area, you'll have to use another provider, but hopefully this tutorial will still show you the basics of how using an online provider works.

Before we continue, we should mention one peculiarity of Tropo's web API (Application Programming Interface). All of these web providers will make HTTP requests to your application in order to deliver incoming messages to you. Tropo also has to make a request to your application when you want to send a message. For that to work, your application has to make a call first to Tropo, asking Tropo to call your app, so that then you can send a message. The

Tropo backend for RapidSMS handles all that for you, so you don't need to worry about in when things are working, but that'll be good to know if you need to debug it when it's not working.

Get an account

To create a Tropo account, go to https://www.tropo.com/account/register.jsp and fill in the form.

Create an app at Tropo

Go to https://www.tropo.com/applications/ and create a new WebAPI application. We'll refer to this application from here on as your *Tropo app*, to distinguish it from your RapidSMS apps.

Configure it as follows:

Tropo WebAPI Application Name: Anything you like; this only appears on the Tropo site and is not needed in your RapidSMS app.

What URL powers voice calls to your app? We don't need this, but it cannot be blank. We recommend just copying the messaging URL you enter into the next field.

What URL powers SMS/messaging calls to your app? This is a URL that Tropo will make requests to when interacting with your RapidSMS app, as we mentioned before. You can use something like `https://yourhost.example.com/tropo`. This needs to correspond to a URL configuration in your RapidSMS app. We'll talk more about this when we get to configuration.

Phone Numbers: You'll need a `Voice & Messaging` phone number. Your app will receive text messages at this number, and will use this number as the source number when sending messages. Click `Add a new phone number` to add a number. After adding this number, make a note of it.

You can ignore the other phone numbers.

Outbound tokens:

Voice: You can ignore this token.

Messaging: Click on this token string to display a popup window where you can copy the entire token. Save it for later. Click the close button in the upper right of the window.

Click the `Update Application` button to save your settings.

Install the backend

Add the Tropo RapidSMS backend to your requirements by editing `requirements/base.txt`:

```
Django>=1.5,<1.6
RapidSMS==0.14.0
South==0.7.6
rapidsms-tropo>=0.2.0
```

Then use pip to install it:

```
$ pip install -r requirements/base.txt
```

That will pull in rapidsms-tropo, along with its dependencies.

Configure RapidSMS and the backend

You'll need to add or change a few settings in your application.

INSTALLED_APPS: Add "rtropo" to INSTALLED_APPS.

INSTALLED_BACKENDS: Add a new entry to *INSTALLED_BACKENDS* for the Tropo backend to talk to your Tropo account. It will look something like this:

```
INSTALLED_BACKENDS = {
    ...,
    "my-tropo-backend": {
        "ENGINE": "rtropo.outgoing.TropoBackend",
        'config': {
            # Your Tropo application's outbound token for messaging
            'messaging_token': '(some long hex string)',
            # Your Tropo application's voice/messaging phone number (including country code)
            'number': '+1-555-555-1212',
        },
    },
}
```

URLs: Tropo will be making HTTP requests to your RapidSMS site, so you'll need to configure a URL for it to call. Edit your site's top-level urls.py file, and add a URL definition for the messaging URL that you configured in your Tropo app on the Tropo site. It should call the Tropo backend's view for receiving messages (`rtropo.views.message_received`), and pass the name of the backend you used in *INSTALLED_BACKENDS*. The URL pattern should match the URL you configured at Tropo. For example, if you configured the URL `https://yourhost.example.com/tropo/` in your Tropo app, then configure a Django URL like this:

```
from rtropo.views import message_received

urlpatterns = patterns('',
    ...,
    url(r'^tropo/',
        message_received,
        kwargs={'backend_name': 'my-tropo-backend'}),
    ...
)
```

Try it out

Start your site. Get out your cell phone, and send a text message to your phone number at Tropo. Send "ping" and you should get back "pong", if the application we added in part 1 is still configured.

Troubleshooting

If you don't get a response, first check your application's logs for errors and if you find any, follow them up. If you don't find any, or you fix them and try again and still don't get a response, then you'll want to methodically work through the steps your message and its response have to take and check things out.

Did Tropo get your text?

Tropo has an excellent debugging tool. When you're logged in to their site, you'll see a link near the top right, "Application Debugger". Follow that link and you'll see a window which will show voluminous logging information.

The window starts out empty, so once you have it open, send a new message to your Tropo number and see what shows up. If nothing does, then Tropo didn't get your message. Go back to your application settings on the Tropo site and check the phone number again, then double-check you're not misdialing it when you send the message.

Did Tropo call your site?

We should be able to tell from the logs in the Tropo application debugger what Tropo did with the message. The window automatically scrolls to the end, so scroll back up to the top. Then start scanning the log messages.

Hopefully after 10 or 20 messages have gone by, you'll see something like this:

> #TROPO#: Found hostedCloudDnsApplicationInfo [_url=https://hostname.example.com/tropo/, _type=tropo-web, _account=NNNNN, _userName=XXXXXX, _appId=NNNNN, _odf=cusd, _serviceId=NNNNNN, _platform=NNN][endpoint=NNNNNNNNN]

That tells you that Tropo matched the incoming message to your application. Double-check the URL there.

Was Tropo's call to your site successful?

Keep scanning down the logs, paying particular attention to lines with your URL in them, and you should eventually find Tropo sending a request to your application. It might look like this:

> #TROPO#: Sending TropoML Payload on Tropo-Thread-3b43948e921da539a358747c389567a8 [url=http://host.example.com/tropo/]: {"session":{"id":"3b43948e921da539a358747c389567a8","accountId":"NNNNN","timest
> 05-17T15:44:08.724Z","userType":"HUMAN","initialText":"MYMESSAGE","callId":"(hex string)","to":{"id":"15555551212","name":null,"channel":"TEXT","network":"SMS"},"from":{"id":"15555551212","name":nul
> whole lot of SIP headers omitted here}}}

If the application failed to handle the request, that might be followed shortly by something like this:

> #TROPO#: Received non-2XX status code on Tropo-Thread-163cd6755723938b4b19003576b16212 [url=http://home.example.com/tropo/, code=500]

That indicates that the request got a response status code of 500 from your app. If you see this, you'll have to go back to your app and add more logging or find another way to determine what's going wrong when Tropo calls your app.

What you'd like to see instead would be a log message like this:

> #TROPO#: Received new TropoML document on Tropo-Thread-5312f2c74f36e1421622564e18c1c297: {"tropo": [{"hangup": {}}]}

That shows the rapidsms-tropo backend responded to Tropo with a little Tropo program, as it should.

Did your site call Tropo back?

In order to send a response, your site has to make a call to Tropo, then Tropo calls your site back, and finally your site responds to that request with the command to send the response message. (This convoluted workflow seems to be unique to Tropo; with most other providers, your site just calls the provider and sends the command to send a message.)

This will all show up in the debug log as well. To confuse the issue, this flow might overlap with the previous flow - your site might call Tropo while still in the middle of handling the request from Tropo. However, you can distinguish the two calls by looking at the `SessionID` column in the debugger. The first part of that is just the line number in the log window, but the second part identifies the session, and will be different on the messages associated with a different call.

Here's a message indicating your site has called Tropo:

> #TROPO#: HTTPDriver.doGet(): action = create

And further down with the same session ID, you should see another message showing Tropo calling your app again:

> #TROPO#: Sending TropoML Payload on Tropo-Thread-5acf02a5867a557bd6b31212f47a5c56 [url=http://home.example.com:9123/tropo/]: {"session":{"id":"5acf02a5867a557bd6b31212f47a5c56","accountId":"NNNNN","
> 05-17T16:54:54.307Z","userType":"NONE","initialText":null,"callId":null,"parameters":{(contents
> omitted)}}}

Keep looking for the same session ID to see if this was successful. Eventually you should see something like:

> #TROPO#: Received new TropoML document on Tropo-Thread-5acf02a5867a557bd6b31212f47a5c56: {"tropo": [{"message": {"to": "15555551212", "say": {"value": "Sorry, RapidSMS could not understand your message."}, "from": "+15555551212", "network": "SMS", "channel": "TEXT"}}]}

This is the rapidsms-tropo backend telling Tropo to send a message "Sorry, RapidSMS could not understand your message.".

Did Tropo send the response message?

Continue following the log messages for the same session. Searching for the text of the response message might be helpful. You're looking for a log message showing Tropo delivering the message externally. It might look like this:

> #MRCP#: (o)ANNOUNCE rtsp://10.6.69.204:10074/synthesizer/ RTSP/1.0rnCseq: 3rnSession: 1368809694451-15745b70-b9b143c0-00000585rnContent-Type: application/mrcprnContent-Length: 397rnrnSPEAK 141650001 MRCP/1.0rnKill-On-Barge-In: falsernSpeech-Language: imrnVendor-Specific-Parameters: IMified-Network=SMS;IMified-From=+15555551212;IMified-Bot-Key=88A17A15-CCC1-404B-806434AD47E4B442;IMified-User=tel:+15555551212rnContent-Type: application/synthesis+ssmlrnContent-Length: 103rnrn<?xml version="1.0" encoding="UTF-8"?><speak>Sorry, RapidSMS could not understand your message.</speak> #[1368809694451-15745b70-b9b143c0-00000585][10.6.69.204:10074][10.6.69.204:59469][4602a1bcfe5482f8b25066886e8a7496][456902][77104]

Most of that we can ignore, bug we should see our phone numbers and the text message. After that, we should see another log message showing the response, hopefully successful:

> #MRCP#: (i)RTSP/1.0 200 OKrnSession: 1368809694451-15745b70-b9b143c0-00000585rnCseq: 3rnContent-Type: application/mrcprnContent-Length: 38rnrnMRCP/1.0 141650001 200 IN-PROGRESSrnrn #[1368809694451-15745b70-b9b143c0-00000585][10.6.69.204:10074][10.6.69.204:59469][4602a1bcfe5482f8b25066886e8a7496][456902][77104]

Again, we can ignore most of that, but "200 OK" is a good sign.

Next steps

Continue reading the documentation. There's a lot of useful information. Some of it you might want to skim for now, but it'll give you an idea of what RapidSMS can do, and where to look for more details when you're ready to try new things.

CHAPTER 4

Getting help

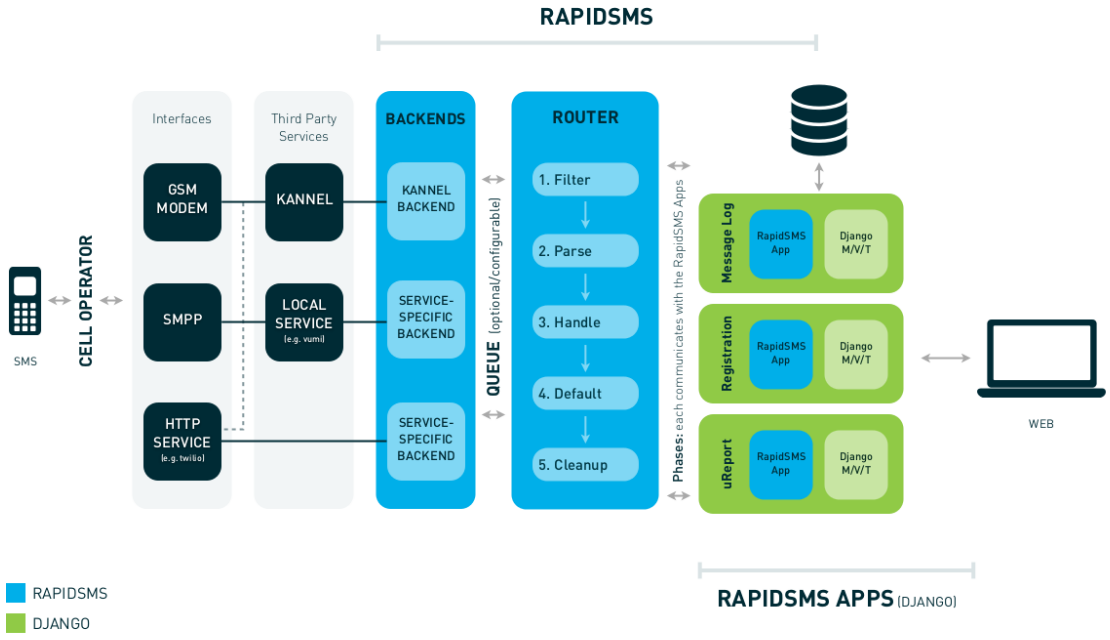
If you need help while using RapidSMS, there are several ways you can get it.

- Ask on the #rapidsms IRC channel on Freenode. You can use the webchat client for this too. The #rapidsms IRC channel, like most IRC channels, is pretty informal. You don't need to ask whether it's okay to ask a question - just ask it.
- Ask on the rapidsms mailing list. This list is used to ask and answer questions, help work through bugs, and discuss general RapidSMS topics.

  Note that there's a second mailing list, `rapidsms-dev`. You're welcome to join both, but `rapidsms-dev` is intended for discussing development of RapidSMS itself (changing the RapidSMS code), while `rapidsms` is to talk about using RapidSMS, writing apps that use RapidSMS, etc.

CHAPTER 5

RapidSMS Architecture Overview



You can also view the full-sized version.

5.1 Introduction

RapidSMS is divided into a few core components:

- *Applications*
- *Backends*
- *Router*

If you are new to RapidSMS most likely you will want to develop Applications.

5.2 Applications

RapidSMS applications, or "apps", perform one or more of the following functions:

- Performs your business logic
- Handles and responds to messages
- Extends the data model
- Creates a web interface with Django views and templates

For example, a registration application may provide a messaging protocol for users to register themselves into the system. In general, you'll probably be writing applications more than anything else. Please see the application documentation for more information.

RapidSMS represents the entities it communicates with using *Contacts*, which you'll also want to understand before writing applications.

5.3 Backends

Backends receive messages from external sources and deliver messages from applications to external sources. Example backends include:

- Using Kannel to communicate to a GSM modem connected over USB or Serial
- Using Twilio or Clickatell to send and receive SMS messages over HTTP

Please see the backend documentation for more information.

5.4 Router

The router is the message processing component of RapidSMS. It provides the infrastructure to receive incoming, send outgoing messages, and gluing together your applications and backends. RapidSMS provides several built-in routers to use based on the needs of your application.

Please see the router documentation for more information.

CHAPTER 6

RapidSMS Applications

RapidSMS applications are Django apps which contain custom logic for processing incoming and outgoing messages. When the router receives an incoming or outgoing message, it triggers a series of phases through which its associated applications can process the message. Any number of RapidSMS applications can be used in a project.

Each RapidSMS application defines a class that extends from `rapidsms.apps.base.AppBase`, kept in the `app.py` submodule of a Django app. The Django app also contains models, views, and methods required by the application.

As an example, we might create a simple application that replies 'pong' after receiving the message 'ping':

```
# In pingpongapp/app.py

from rapidsms.apps.base import AppBase


class PingPong(AppBase):

    def handle(self, msg):
        if msg.text == 'ping':
            msg.respond('pong')
            return True
        return False
```

After associating the PingPong application with the router, new incoming and outgoing messages received by the router are passed through the application for processing. All incoming 'ping' messages will receive a 'pong' reply. In general, the `send` and `receive` methods in the messaging api abstract the logic needed for passing messages to the router.

Application and router behavior in RapidSMS are intertwined. In this section, we focus on the behavior specific to applications, with references to some key areas where this behavior is tied to the router. For more information about routing messages through applications, see the router documentation.

6.1 Application Structure

A RapidSMS application is contained in a Django app. Each application defines a class that extends from `rapidsms.apps.base.AppBase`, kept in the `app.py` submodule of the Django app.

The router maintains a collection of associated applications through which to route incoming and outgoing messages. *Application discovery* is managed through the `BaseRouter.add_app` method. The default router, `BlockingRouter`, loads applications upon initialization by calling `BaseRouter.add_app` on each app listed in the optional `apps` argument or in `INSTALLED_APPS`.

6.2 Incoming Message Processing

Note: See also the *router documentation on incoming message processing*.

The router receives each incoming message through its `incoming` method. In `BaseRouter.receive_incoming`, the message is passed sequentially to the router's associated applications in each of five processing phases. Applications provide the code to execute each phase. The router provides hooks which allow an application to filter out a message, skip phases, or stop further processing.

Important: The order in which the router chooses applications to process messages is extremely important, because each application will have the opportunity to block subsequent applications from processing a message.

The logic for each phase is defined in a method of the same name in the `AppBase` class. By default, no action is taken at any phase. Each subclass may choose to override any of the default methods to use custom logic on incoming messages.

1. *filter* - **Optionally abort further processing of the incoming message.** The *filter* phase is executed before any other processing or modification of the incoming message. If an application returns `True` from this phase, the message is filtered out and no further processing will be done by any application (not even *cleanup*).

   Example: An application that filters out spam messages:

```
from rapidsms.apps.base import AppBase

class SpamFilter(AppBase):

    def filter(self, msg):
        """Filter out spam messages."""
        if msg.text == "Congratulations, you've won a free iPod!":
            return True  # This message is probably spam and should not be
                         # processed any more.
        return False
```

2. *parse* - **Modify message in a way that is globally useful.** This phase is used to modify the incoming message in a way that could be useful to other applications. All messages that aren't filtered go through the *parse* phase of every application. No INSERTs or UPDATEs should be done during this phase.

   Example: An application adds metadata about phone number registration to each message.

3. *handle* - **Respond to the incoming message.** The router passes incoming messages through the *handle* phase of each application until one of them returns `True`. All subsequent apps will not handle the message.

   It is considered best practice to return `True` during the *handle* phase if the application responds to or otherwise alters the message. Although an application may return `False` in order to allow other applications to handle the message, remember that the *default* phase will execute if no application returns `True` during *handle*.

   As mentioned above, the order in which the router chooses to send messages to applications is very important. For example, you may wish to have 'keyword' applications (which look for a specific trigger word) handle a message before more general applications that use a regex to match possible text.

4. *default* - **Execute a default action if no application returns True during the handle phase.** For example, an application might want to provide additional help text or a generic response if no other application has handled the message. The application can return `True` from this method in order to prevent the remaining applications from executing their *default* stage.

5. *cleanup* - **Clean up work from previous phases.**

6.3 Outgoing Message Processing

Note: See also the *router documentation on outgoing message processing*.

The router receives each outgoing message through its `outgoing` method. Messages are processed in a manner similar to incoming messages, except only one phase, *outgoing*, is defined. In `BaseRouter.send_outgoing`, the message is processed sequentially by the router's associated applications. However, the applications are called in reverse order with respect to the order they are called in `BaseRouter.receive_incoming`, so the first application called to process an incoming message is the last application that is called to process an outgoing message. If any application returns `False` during the *outgoing* phase, all further processing of the message will be aborted.

The logic for the *outgoing* phase is defined in a method of the same name in the `AppBase` class. By default, no action is taken during this phase. Each subclass may choose to override the default method to use custom logic on outgoing messages.

6.4 Router Events: `start` and `stop`

For historical reasons, each application can provide start-up and shut-down logic in the `start` and `stop` methods, respectively. These methods are called from `BaseRouter` when the router is started or stopped. However, this behavior has never been enforced. A "stopped" router can still receive messages and will route them to applications, even "stopped" applications. As we move toward v1.0, we expect to remove these methods from `BaseApp`.

6.5 Scheduling tasks

If your application needs to run tasks asynchronously, either on-demand or on a schedule, you can of course use any mechanism that works in Django. The RapidSMS project recommends using Celery, and there are some advantages to using Celery in RapidSMS applications compared to other schedulers. See Using Celery for Scheduling Tasks

6.6 Contrib and Community Applications

There are many existing RapidSMS applications. The applications in `rapidsms.contrib` are maintained by core developers and provide broad-reaching functionality that will be useful to many developers. We also provide a directory of community-maintained RapidSMS applications that may be useful in your project.

6.6.1 RapidSMS Apps

One day soon, all usable RapidSMS apps will be found or linked to from the following repositories:

- http://github.com/rapidsms/rapidsms-contrib-apps-dev (optional apps, restricted committers)
- http://github.com/rapidsms/rapidsms-community-apps-dev (optional apps, open committers)
- http://github.com/nyaruka/rapidsms-xforms (interactive form builder)

Community Apps

This is a list of apps that are known to be currently in development, but exist in other repositories on github:

- An application describing common models for the health domain: http://github.com/unicefuganda/rapidsms-healthmodels
- Gelvin and KSam's Generic Django app for using the Entity-Attribute-Value design pattern: http://github.com/mvpdev/django-eav
- Nic's xforms app, A way of building data collection forms on the fly, exposing them to both an SMS parser and any ODK client: http://github.com/nyaruka/rapidsms-xforms
- Nic's app for processing messages in the http thread: http://github.com/nyaruka/rapidsms-httprouter
- Caktus' app for processing messages in an http process with Celery: http://github.com/caktus/rapidsms-threadless-router
- An app for creating simple-question, simple-response (i.e. trainingless) SMS campaigns: http://github.com/unicefuganda/rapidsms-polls
- An app for asking a series of questions (for example, to register a user in the system with various contact information): http://github.com/unicefuganda/rapidsms-script
- An alternative to djtables, this app lets you build list admin views, maps, charts, and customizable dashboards from * generic view templates: http://github.com/unicefuganda/rapidsms-generic

All of these are in various states of development, however most have been in production for some time. If you have an interest in contributing to any of these apps, please post to the google group!

CHAPTER 7

RapidSMS Backends

Backends define how RapidSMS communicates with the outside world. The router uses backends to send and receive messages and all text messages will eventually pass through a backend. They handle a two-way messaging protocol:

- **Incoming messages:** Messages received by RapidSMS from an external source. All incoming messages are received over HTTP and processed by a Django view. Once received, backends will pass messages to the router for processing.
- **Outgoing messages:** Messages sent by RapidSMS to an external source. The router will pass messages to backends once processed. RapidSMS sends messages over HTTP.

7.1 Setting up RapidSMS with Kannel

Kannel is a free and open source SMS gateway that can be configured for use with RapidSMS. While in-depth Kannel configuration is outside the scope of this documentation, it's possible to configure Kannel to connect directly to USB or serial GSM modems as well as third party HTTP or SMPP gateways. For more information about the connections Kannel supports (what Kannel calls an "SMS Center" or "SMSC"), see the in-depth Kannel user guide and refer to "Chapter 6. Setting up a SMS Gateway".

The following guide will help you setup Kannel on Ubuntu to talk to a single GSM modem and RapidSMS installation.

7.1.1 Installing Kannel

A `kannel` package is included with Ubuntu, so installation is very easy:

```
sudo apt-get install kannel
```

By default in Ubuntu, Kannel starts a WAP gateway but does not start the SMS gateway. To change this behavior, first stop Kannel:

```
sudo service kannel stop
```

Now, edit `/etc/default/kannel` and uncomment the line starting with `START_SMSBOX`. If you won't be using the WAP gateway (if you don't know what that is you probably won't be), you can also disable it by commenting out `START_WAPBOX=1`. **Note:** Simply setting START_WAPBOX=0 will not disable it; you must comment out the line:

```
sudo vim /etc/default/kannel # or use your favorite editor
```

Finally, start Kannel again (note it will say "Starting WAP gateway" even if it's only starting the SMS gateway):

```
sudo service kannel start
```

You can check that it's running by looking at `ps ax | grep kannel`. You should see something like this:

```
2446 ?        Ss     0:00 /usr/sbin/run_kannel_box --pidfile /var/run/kannel/kannel_bearerbox.pid --r
2447 ?        Sl     0:00 /usr/sbin/bearerbox -v 4 -- /etc/kannel/kannel.conf
2460 ?        Ss     0:00 /usr/sbin/run_kannel_box --pidfile /var/run/kannel/kannel_smsbox.pid --no-e
```

7.1.2 Setting up the fake SMSC for testing

Kannel includes support for a Fake SMSC which can be useful during setup for testing both Kannel and RapidSMS. The relevant utility is included in the `kannel-extras` package:

```
sudo apt-get install kannel-extras
```

To make things simpler, we'll first setup Kannel and RapidSMS to work with a Fake SMSC, and then attempt to connect it to a USB modem.

Configuring Kannel for the first time

The easiest way to get Kannel working with RapidSMS is to start with a sample Kannel configuration. To get started, copy and paste the following into `/etc/kannel/kannel.conf`, replacing everything currently in the file (make a backup first if you'd like):

```
# CONFIGURATION FOR USING SMS KANNEL WITH RAPIDSMS
#
# For any modifications to this file, see Kannel User Guide
# If that does not help, see Kannel web page (http://www.kannel.org) and
# various online help and mailing list archives
#
# Notes on those who base their configuration on this:
#  1) check security issues! (allowed IPs, passwords and ports)
#  2) groups cannot have empty rows inside them!
#  3) read the user guide

include = "/etc/kannel/modems.conf"

#------------------------------------------
# CORE
#
# There is only one core group and it sets all basic settings
# of the bearerbox (and system). You should take extra notes on
# configuration variables like 'store-file' (or 'store-dir'),
# 'admin-allow-ip' and 'access.log'

group = core
admin-port = 13000
smsbox-port = 13001
admin-password = CHANGE-ME
status-password = CHANGE-ME
admin-deny-ip = "*.*.*.*"
admin-allow-ip = "127.0.0.1"
box-deny-ip = "*.*.*.*"
box-allow-ip = "127.0.0.1"
log-file = "/var/log/kannel/bearerbox.log"
log-level = 0
```

```
#-------------------------------------------
# SMSC CONNECTIONS
#
# SMSC connections are created in bearerbox and they handle SMSC specific
# protocol and message relying. You need these to actually receive and send
# messages to handset, but can use GSM modems as virtual SMSCs

# Here is a sample SMSC for use with the /usr/lib/kannel/test/fakesmsc command

group = smsc
smsc = fake
smsc-id = FAKE
port = 10000
connect-allow-ip = 127.0.0.1

#-------------------------------------------
# SMSBOX SETUP
#
# Smsbox(es) do higher-level SMS handling after they have been received from
# SMS centers by bearerbox, or before they are given to bearerbox for delivery

group = smsbox
bearerbox-host = 127.0.0.1
sendsms-port = 13013
sendsms-chars = "0123456789 +-"
log-file = "/var/log/kannel/smsbox.log"
log-level = 0
access-log = "/var/log/kannel/smsbox-access.log"
reply-couldnotfetch = "Your message could not be processed at this time.  Please try again later. (er
reply-requestfailed = "Your message could not be processed at this time.  Please try again later. (er
reply-couldnotrepresent = "Your message could not be processed at this time.  Please try again later.
http-request-retry = 3
http-queue-delay = 10

# SEND-SMS USERS
#
# These users are used when Kannel smsbox sendsms interface is used to
# send PUSH sms messages, i.e. calling URL like
# http://kannel.machine:13013/cgi-bin/sendsms?username=tester&password=foobar...

# This is the username and password that RapidSMS uses to deliver SMSes to
# Kannel.  It must also set the 'smsc' variable in the query string, so that
# Kannel knows which SMSC to use to route the message.

group = sendsms-user
username = rapidsms
password = CHANGE-ME
user-deny-ip = "*.*.*.*"
user-allow-ip = "127.0.0.1;"

#-------------------------------------------
# SERVICES
#
# These are 'responses' to sms PULL messages, i.e. messages arriving from
# handsets. The response is based on message content. Only one sms-service is
# applied, using the first one to match.

# The 'ping-kannel' service let's you check to see if Kannel is running,
```

```
# even if RapidSMS is offline for some reason.

group = sms-service
keyword = ping-kannel
text = "Kannel is online and responding to messages."

# There should be always a 'default' service. This service is used when no
# other 'sms-service' is applied.  These relay incoming messages from any
# configured SMSCs to the appropriate HTTP backend URLs in RapidSMS.
# By setting 'accepted-smsc', we are assured that messages are routed to
# the appropriate backend in RapidSMS.

group = sms-service
keyword = default
catch-all = yes
accepted-smsc = FAKE
# don't send a reply here (it'll come through sendsms):
max-messages = 0
get-url = http://127.0.0.1:8000/backend/kannel-fake-smsc/?id=%p&text=%a&charset=%C&coding=%c
```

You'll notice the file includes a file called `modems.conf` at the top. You can copy this file from where Ubuntu installed it as follows:

```
sudo cp /usr/share/doc/kannel/examples/modems.conf /etc/kannel/
```

Next, restart Kannel to reload the new configuration:

```
sudo service kannel restart
```

When you look at the process list (`ps ax | grep kannel`), you should see a 4th process for the smsbox now started, like so:

```
3231 ?        Ss     0:00 /usr/sbin/run_kannel_box --pidfile /var/run/kannel/kannel_bearerbox.pid --r
3232 ?        Sl     0:00 /usr/sbin/bearerbox -v 4 -- /etc/kannel/kannel.conf
3243 ?        Ss     0:00 /usr/sbin/run_kannel_box --pidfile /var/run/kannel/kannel_smsbox.pid --no-e
3245 ?        Sl     0:00 /usr/sbin/smsbox -v 4 -- /etc/kannel/kannel.conf
```

You can further test that Kannel is running by using the fake SMSC (used only for testing) to use the "ping-kannel" service that we included in the Kannel configuration above:

```
/usr/lib/kannel/test/fakesmsc -m 1 "123 789 text ping-kannel"
```

On the last line of the output you should see the message that was sent by the `ping-kannel` service, e.g.:

```
INFO: Got message 1: <789 123 text Kannel is online and responding to messages.>
```

Press Control-C to kill the `fakesmsc` command and return to the prompt.

Adding a backend for the fake SMSC to RapidSMS

Now that Kannel is installed and configured correctly, adding support for the Kannel backend to your existing RapidSMS project is not difficult. To begin, simply add the following to your existing `INSTALLED_BACKENDS` configuration in your `settings.py` file:

```
INSTALLED_BACKENDS = {
    # ...
    # other backends, if any
    "kannel-fake-smsc" : {
        "ENGINE":  "rapidsms.backends.kannel.KannelBackend",
```

```
        "sendsms_url": "http://127.0.0.1:13013/cgi-bin/sendsms",
        "sendsms_params": {"smsc": "FAKE",
                           "from": "123", # not set automatically by SMSC
                           "username": "rapidsms",
                           "password": "CHANGE-ME"}, # or set in localsettings.py
        "coding": 0,
        "charset": "ascii",
        "encode_errors": "ignore", # strip out unknown (unicode) characters
    },
}
```

"host" and "port" should no longer be included in the backend configuration. Next, you need to add an endpoint to your urls.py for the newly created backend. You can do this like so:

```
from django.conf.urls.defaults import *
from rapidsms.backends.kannel.views import KannelBackendView

urlpatterns = patterns('',
    # ...
    url(r"^backend/kannel-fake-smsc/$",
        KannelBackendView.as_view(backend_name="kannel-fake-smsc")),
)
```

You can make the Django URL pattern whatever you like, but the convention is to make it backend/ followed by the name of your backend (from the settings file) and a final /.

Now, you should be able to start RapidSMS like so:

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

And test connection using the echo app in RapidSMS (if installed in your project):

```
/usr/lib/kannel/test/fakesmsc -m 1 "123 789 text echo hi"
```

You should see the message get echoed back to you on the last line:

```
INFO: Got message 1: <123 123 text hi>
```

7.1.3 Adding support for a GSM Modem SMSC

This section assumes that you've already installed, configured, and setup Kannel to use the Fake SMSC as described above. Once you have Kannel and RapidSMS configured, adding support for additional SMSCs (such as a GSM modem) is fairly easy. It also assumes that you already have a GSM modem connected to your computer, and that you know the device location (e.g., /dev/ttyUSB0) of that modem.

Adding the GSM modem to the Kannel configuration

Using the base configuration given above, add the following to the section titled "SMSC CONNECTIONS" in /etc/kannel/kannel.conf, changing the device = /dev/ttyUSB0 line so that it points to the right device:

```
group = smsc
smsc = at
smsc-id = usb0-modem
my-number = 1234
modemtype = auto
device = /dev/ttyUSB0
```

Next, add the following `sms-service` at the end of the file, which will send incoming messages from the modem to RapidSMS via HTTP:

```
group = sms-service
keyword = default
catch-all = yes
accepted-smsc = usb0-modem
# don't send a reply here (it'll come through sendsms):
max-messages = 0
get-url = http://127.0.0.1:8000/backend/kannel-usb0-smsc/?id=%p&text=%a&charset=%C&coding=%c
```

Make sure to restart Kannel to reload the configuration:

```
sudo service kannel restart
```

Adding a backend for the GSM modem to RapidSMS

Finally, add a second Kannel backend to your `settings.py` which will setup the necessary router infrastructure to send and receive messages via the USB modem you configured above in Kannel:

```
INSTALLED_BACKENDS = {
    # ...
    "kannel-usb0-smsc" : {
        "ENGINE":  "rapidsms.backends.kannel.KannelBackend",
        "sendsms_url": "http://127.0.0.1:13013/cgi-bin/sendsms",
        "sendsms_params": {"smsc": "usb0-modem",
                           "from": "+SIMphonenumber", # not set automatically by SMSC
                           "username": "rapidsms",
                           "password": "CHANGE-ME"}, # or set in localsettings.py
        "coding": 0,
        "charset": "ascii",
        "encode_errors": "ignore", # strip out unknown (unicode) characters
    },
}
```

`"host"` and `"port"` should no longer be included in the backend configuration. Next, you need to add an endpoint to your `urls.py` for the newly created backend. You can do this like so:

```
from django.conf.urls.defaults import *
from rapidsms.backends.kannel.views import KannelBackendView

urlpatterns = patterns('',
    # ...
    url(r"^backend/kannel-usb0-smsc/$",
        KannelBackendView.as_view(backend_name="kannel-usb0-smsc")),
)
```

Now, the next time you call `./manage.py runserver`, Django should begin processing requests for both the Kannel backends that you created (one for the fake SMSC and one for the GSM modem).

7.1.4 Delivery Report

RapidSMS can take advantage of Kannel's SMS Delivery Report functionality. This is useful if you'd like to track the status of a message after it's been passed to Kannel for processing. Kannel will use a callback URL to notify us. Enabling this feature will allow you to view delivery reports, for each message, in the Django admin.

1. Add `rapidsms.backends.kannel` to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # Other apps here
    "rapidsms.backends.kannel",
)
```

2. Add `kannel/` URLs to your urlconf:

```
urlpatterns = patterns("",
    # ...
    url(r'^kannel/', include('rapidsms.backends.kannel.urls')),
)
```

3. Add the necessary database tables (omit `--migrate` if you're not using South):

```
    python manage.py syncdb --migrate
```

4. Update your Kannel backend settings with `delivery_report_url`. This is the URL Kannel will use to notify RapidSMS. Kannel requires a full URL, including the protocol and authority, even if you're only communicating locally. RapidSMS will automatically append the necessary path and query string arguments, so you only need to include the protocol and authority information, such as `http://127.0.0.1:8000` or `http://example.com`. Our example is local:

```
INSTALLED_BACKENDS = {
    # ...
    "kannel-usb0-smsc" : {
        "ENGINE":  "rapidsms.backends.kannel.KannelBackend",
        "sendsms_url": "http://127.0.0.1:13013/cgi-bin/sendsms",
        "sendsms_params": {"smsc": "usb0-modem",
                           "from": "+SIMphonenumber", # not set automatically by SMSC
                           "username": "rapidsms",
                           "password": "CHANGE-ME"}, # or set in localsettings.py
        "coding": 0,
        "charset": "ascii",
        "encode_errors": "ignore", # strip out unknown (unicode) characters
        "delivery_report_url": "http://127.0.0.1:8000",
    },
}
```

You can view delivery reports in the Django admin.

7.1.5 Troubleshooting

For help troubleshooting, please carefully review the relevant log files in `/var/log/kannel` as well as the output of the `./manage.py runserver` command. For additional help configuring Kannel, review the Kannel user guide or subscribe to the Kannel users mailing list.

7.2 Setting up RapidSMS with Vumi

Vumi is a free and open source super scalable messaging platform written in Python. Vumi can connect to third party gateways via protocols like HTTP and SMPP. Please read Vumi's documentation for additional information.

The following guide will help you setup Vumi on Ubuntu to talk to a SMPP simulator and RapidSMS installation.

7.2.1 Using an SMPP simulator

For local development, it's easy to setup an SMPP simulator for testing purposes. Vumi suggests using SMPPSim. SMPPSim is a testing utility which mimics the behavior of the Short Message Peer to Peer Protocol (SMPP) based Short Message Service Center (SMSC).

Note: SMPPSim requires Java. Please install Java before proceeding with these instructions.

To download SMPPSim, run the following commands:

```
wget http://www.seleniumsoftware.com/downloads/SMPPSim.tar.gz
tar zxf SMPPSim.tar.gz
```

SMPPSim runs on port 80 by default. Let's set this to a higher port to make it easier to run without superuser privileges. Open `conf/smppsim.props` and change the `HTTP_PORT` line to:

```
HTTP_PORT=8080
```

You can use the provided shell scripts to start SMPPSim:

```
chmod +x startsmppsim.sh
./startsmppsim.sh
```

Now visit http://localhost:8080 to use SMPPSim's web interface. You'll use this interface to create mobile-originated (MO) messages to send to Vumi.

7.2.2 Installing and setting up Vumi for the first time

Note: As of this writing, the RapidSMS/Vumi integration is planned for merge into an official Vumi release, but currently resides in the `develop` Vumi branch. When complete, we will update this documentation accordingly.

Clone the Vumi GitHub repository:

```
git clone git@github.com:praekelt/vumi.git
cd vumi
git checkout develop
```

Install Vumi's Python dependencies:

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

Setup the proper RabbitMQ user/vhost using the provided utility script:

```
sudo ./utils/rabbitmq.setup.sh
```

Create `config/rapidsms.yaml` using the following configuration:

```
smpp_transport:
    transport_name: "transport"
    system_id: smppclient1  # username
    password: password      # password
    host: localhost         # the host to connect to
    port: 2775              # the port to connect to

rapidsms_relay:
    transport_name: 'transport'
    rapidsms_url: "http://127.0.0.1:8000/backend/vumi-fake-smsc/"
```

```
    web_path: "/send/"
    web_port: 9000
    send_to:
        default:
            transport_name: 'transport'
            from_addr: '1234' # not set automatically by SMSC

workers:
    smpp_transport: vumi.transports.smpp.SmppTransport
    rapidsms_relay: vumi.application.rapidsms_relay.RapidSMSRelay
```

This configures a Vumi `SmppTransport` to communicate to *SMPPSim* and a Vumi `RapidSMSRelay` to communicate to RapidSMS. While not required for this setup, you'll need to set `from_addr` to your phone number if using a real SMSC.

Now we can start Vumi using our config file:

```
twistd -n start_worker --worker-class vumi.multiworker.MultiWorker --config config/rapidsms.yaml
```

7.2.3 Adding a backend for the fake SMSC to RapidSMS

Now that Vumi is installed and configured correctly, adding support for the Vumi backend to your existing RapidSMS project is not difficult. To begin, simply add the following to your existing *INSTALLED_BACKENDS*:

```
INSTALLED_BACKENDS = {
    # ...
    # other backends, if any
    "vumi-fake-smsc": {
        "ENGINE":  "rapidsms.backends.vumi.VumiBackend",
        "sendsms_url": "http://127.0.0.1:9000/send/",
    },
}
```

Next, you need to add an endpoint to your `urls.py` for the newly created backend. You can do this like so:

```
from django.conf.urls.defaults import *
from rapidsms.backends.vumi.views import VumiBackendView

urlpatterns = patterns('',
    # ...
    url(r"^backend/vumi-fake-smsc/$",
        VumiBackendView.as_view(backend_name="vumi-fake-smsc")),
)
```

You can make the Django URL pattern whatever you like, but the convention is to make it `backend/` followed by the name of your backend (from the settings file) and a final `/`.

Now, you should be able to start RapidSMS like so:

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

That's it! Now you can use SMPPSim to send mobile-originated (MO) messages through Vumi to RapidSMS.

7.2.4 Authentication

Vumi can be protected with basic authentication. To enable it on the Vumi side, create a `passwords` directive in the `rapidsms_relay` configuration:

```
rapidsms_relay:
    transport_name: 'transport'
    rapidsms_url: "http://127.0.0.1:8000/backend/vumi-fake-smsc/"
    web_path: "/send/"
    web_port: 9000
    send_to:
        default:
            transport_name: 'transport'
            from_addr: '1234' # not set automatically by SMSC
    vumi_username: 'username'
    vumi_password: 'password'
```

Then you can update *INSTALLED_BACKENDS* with `sendsms_user` and `sendsms_pass`:

```
 INSTALLED_BACKENDS = {
     "vumi-fake-smsc": {
         "ENGINE":  "rapidsms.backends.vumi.VumiBackend",
         "sendsms_url": "http://127.0.0.1:9000/send/",
         "sendsms_user": "username",
         "sendsms_pass": "password",
     },
 }
```

7.3 The Database Backend

The database backend has a special purpose, and is primarily used for testing. When the router sends an outgoing message via the database backend, instead of sending a text message, the database backend stores the message in a database table.

The database backend is currently only used with httptester and with the unit test harness. Generally, you will not use the database backend in production deployments.

7.3.1 Configuring

To configure the database backend:

1. Add its class to the *INSTALLED_BACKENDS* setting:

```
 INSTALLED_BACKENDS = {
     ...
     "my-db-backend": {
         "ENGINE": "rapidsms.backends.database.DatabaseBackend",
     },
     ...
 }
```

2. Add its app to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'rapidsms.backends.database',
    ...
]
```

3. Create its database table:

```
./manage.py syncdb
```

If you're using South, you should run migrations:

```
./manage.py migrate
```

No URLs need to be configured, since the database backend cannot receive messages from outside RapidSMS.

7.4 Custom Backends

You can create a custom backend if the supplied backends don't suffice. Since backends handle both inbound and outbound communication, the following section is divided into *Incoming Messages* and *Outgoing Messages*, respectively.

The built-in Vumi and Kannel backends use the methods described below, so you can review the source code to see actual implementation examples.

For a more general backend overview, please see *RapidSMS Backends*.

7.4.1 Incoming Messages

HTTP Backend

RapidSMS provides a base suite of HTTP views and forms to help simplify backend creation in `rapidsms.backends.http`. These are useful for standardizing incoming message handling. They can be extended for use in your own backends. The *HTTP Backend* powers both the *Vumi* and *Kannel* backends.

You can, of course, simply write your own Django views to handle incoming messages if the supplied classes do not provide enough flexibility.

Incoming message life cycle

Typically, when RapidSMS receives a message over HTTP, it's processed in the following way:

1. Data from a text message is received by Django over an HTTP request.
2. The HTTP request is routed through a *Backend URL*.
3. This backend view takes the HTTP request and passes it into a backend form.
4. This backend form cleans the message data and checks its validity.
5. If the message is valid, message data is sent to the router for processing via *`rapidsms.router.receive()`*.
6. An HTTP response is sent to the HTTP request sender with an HTTP status code to indicate that the message was received and passed to the router for processing successfully or that there was an error.

The HTTP response from a backend view does not necessarily indicate that the resulting messages were sent by the router, only that the incoming message was added to the queue for processing.

GenericHttpBackendView

class `rapidsms.backends.http.views.`**`GenericHttpBackendView`**(***kwargs*)
Simple view that allows customization of accepted paramters.

http_method_names = ['get', 'post']
Accepts GET and POST by default.

params = {}
Dictionary that defines mappings to `identity` and `text`.

The simplest type of custom backend is an HTTP backend that needs to accept parameters other than `identity` and `text`. To create such a custom backend, one can subclass the `GenericHttpBackendView` as follows:

```
from rapidsms.backends.http.views import GenericHttpBackendView

class MyBackendView(GenericHttpBackendView):
    params = {
        'identity_name': 'phone',
        'text_name': 'message',
    }
```

The `params` dictionary contains key value pairs that map internal names to the keys used in requests to the backend. In the above example, an HTTP request would provide `phone` and `message` parameters.

An URL pattern for this backend might look like:

```
from project_name.app_name.views import MyBackendView

urlpatterns = patterns('',
    url(r'^backends/mybackend/$',
        MyBackendView.as_view(backend_name='mybackend')),
)
```

A request to this backend might look like the following:

```
>>> import urllib
>>> import urllib2
>>> data = urllib.urlencode({'phone': '1112223333', 'message': 'ping'})
>>> request = urllib2.urlopen('http://localhost:8000/backends/mybackend/', data)
>>> request.code
200
>>> request.read()
'OK'
```

Custom Validation

Another custom backend might necessitate handling more parameters in the request, or validating the incoming data differently. A convenient way to do this validation with Django is with forms:

```
from .forms import ExtraParamsHttpBackendForm
from rapidsms.backends.http.views import GenericHttpBackendView

class ExtraParamsHttpBackendView(GenericHttpBackendView):
    form_class = ExtraParamsHttpBackendForm
```

This example application would have the following forms definition:

```
from django import forms
from rapidsms.backends.http.forms import BaseHttpForm

class ExtraParamsHttpBackendForm(BaseHttpForm):
    extra = forms.TextField()
```

```
    def get_incoming_data(self):
        fields = self.cleaned_data.copy()
        return {'identity': self.cleaned_data['identity_name'],
                'text': self.cleaned_data['text_name'],
                'extra': self.cleaned_data['extra']}
```

This uses RapidSMS's BaseHttpForm:

class rapidsms.backends.http.forms.**BaseHttpForm**(**args*, ***kwargs*)
Helper form for validating incoming messages.

Parameters **backend_name** – (Optional) name of the backend

get_incoming_data()
Return a dictionary containing the connection and text for this message, based on the field names passed to __init__().

Must be implemented by subclasses.

lookup_connections(*identities*)
Simple wrapper to ease connection lookup on child forms.

Data coming into this backend would require an `extra` parameter, which would be passed onto the message queue.

Alternatively, here's an example of a backend form with custom validation:

```
from django import forms
from rapidsms.backends.http.forms import BaseHttpForm

MY_NUMBER = '1231231234'

class OnlyTextMeHttpBackendForm(BaseHttpForm):

    def clean_text_name():
        text_name = self.cleaned_data.get('text_name')
        if text_name != MY_NUMBER:
            raise forms.ValidationError(
                'SMS received from number other than {0}'.format(MY_NUMBER)
            )
        return text_name
```

7.4.2 Outgoing Messages

BackendBase

Similar to *HTTP Backend* for incoming messages, `BackendBase` provides the foundation for outbound functionality. All backends will typically extend this base class. This class will be passed the configuration dictionary defined in *Backend Settings*.

class rapidsms.backends.base.**BackendBase**(*router*, *name*, ***kwargs*)
Base class for outbound backend functionality.

configure(***kwargs*)
Configuration parameters from *INSTALLED_BACKENDS* will be passed here after the router is instantiated. You can override this method to parse your configuration.

classmethod find(*module_name*)
Helper function to import backend classes.

Parameters **module_name** – Dotted Python path to backend class name

Returns Imported class object

model
The model attribute is the RapidSMS model instance with this backend name. A new backend will automatically be created if one doesn't exist upon accessing this attribute.

send(*id_*, *text*, *identities*, *context=None*)
Backend sending logic. The router will call this method for each outbound message. This method must be overridden by sub-classes. Backends typically initiate HTTP requests from within this method.

If multiple `identities` are provided, the message is intended for all recipients.

Any exceptions raised here will be captured and logged by the router. If messages to some identities failed while others succeeded, you can provide that information back to the router by adding a list of the identities which failed in a `failed_identities` parameter on the exception. If you do provide that parameter, then the router should assume that all identities *not* listed in `failed_identities` were successfully sent.

Example

```
def send(self, id_, text, identities, context):
    failures = []
    for identity in identities:
        result = send_my_message(identity, text, context)
        if result == 'failed':
            failures.append(identity)
    if failures:
        msg = '%d messages failed.' % len(failures)
        raise MessageSendingError(msg, failed_identities=failures)
```

Parameters

- **id_** – Message ID
- **text** – Message text
- **identities** – List of identities
- **context** – Optional dictionary with extra context provided by router to backend

7.5 Supplied Backends

RapidSMS includes several backends in core for you to use:

- *Kannel backend*
- *Vumi backend*
- *HTTP backend*
- *Database backend*

However, many other backends exist in the RapidSMS community and can be installed for use in your own project. If you can't find a backend that's suitable for your needs, you can write a *custom backend*.

7.6 Configuration

The instructions below describe how backend configuration works in the general sense. Backends will provide their own installation instructions. If you want to install a specific backend, please follow the backend-specific documentation.

All backends will require the following basic configuration:

7.6.1 INSTALLED_BACKENDS

First, you'll need to add your backend to *INSTALLED_BACKENDS*. This setting is a key/value pairing of **backend name** to a configuration dictionary. For example:

```
INSTALLED_BACKENDS = {
    "my-backend1": {
        "ENGINE": "path.to.BackendClass",
        "example-configuration-option": "Yes",
    },
    "my-backend2": {
        "ENGINE": "path.to.OtherBackendClass",
        "use-special-method": True,
    },
}
```

This examples defines two backends named `my-backend1` and `my-backend2`. The **backend name** can be anything, but it will be used by the router and for matching up with *Backend URLs*. The only required configuration option is `ENGINE`, which is the dotted Python path to the backend class. Additional configuration can be supplied to backends.

7.6.2 URLs

Backends communicate over HTTP and Django uses views to process HTTP requests, so all backends require a Django URL endpoint and view to handle incoming messages. For example:

```
from django.conf.urls import patterns, url
from path.to.backend1 import ExampleBackendView
from path.to.backend2 import OtherBackendView

urlpatterns = patterns('',
    url(r"^backend/my-backend1/$",
        ExampleBackendView.as_view(backend_name="my-backend1")),
    url(r"^backend/my-backend2/$",
        OtherBackendView.as_view(backend_name="my-backend2")),
)
```

This example defines two URLs, one for each backend. You can make the Django URL pattern whatever you like, but the convention is to make it `backend/` followed by the matching **backend name** from INSTALLED_BACKENDS and a final `/`. You must also supply the same **backend name** to the backend view via the `backend_name` keyword argument. This example defines two backends named `my-backend1` and `my-backend2`, matching our example *INSTALLED_BACKENDS* above.

Example URL Configuration

If you learn by example, you can follow these steps and test invoking a received message with a few lines of Python. This example is intended to serve as a simple example of configuring *INSTALLED_BACKENDS* and *Backend URLs*.

1. Include the following in urls.py:

```
from rapidsms.backends.http.views import GenericHttpBackendView

urlpatterns = patterns('',
    url(r'^backends/http-backend/$',
    GenericHttpBackendView.as_view(backend_name='http-backend')),
)
```

2. Include the following in settings.py:

```
INSTALLED_BACKENDS = {
    "http-backend": {
        "ENGINE": "rapidsms.contrib.httptester.backend.HttpTesterCacheBackend",
    },
}
```

3. Now in a Python shell:

```
>>> import urllib
>>> import urllib2
>>> data = urllib.urlencode({'identity': '1112223333', 'text': 'echo hello'})
>>> request = urllib2.urlopen('http://localhost:8000/backends/http-backend/', data)
>>> request.code
200
>>> request.read()
'OK'
```

CHAPTER 8

RapidSMS Routers

The router is the message processing component of RapidSMS. It provides the infrastructure and defines the workflow to receive, process and send text messages. Each RapidSMS project can use only one router, which should be chosen based on the needs of the project.

The basics:

- You may use any router, but only one router can be used per project.
- Each router contains a collection of installed apps and backends.
- All routers will trigger a set of phases for message processing.

Application and router behavior in RapidSMS are intertwined. In this section, we focus on the behavior specific to the router, with references to some key areas where this behavior is tied to applications. For more information about processing messages in applications, see the applications documentation.

8.1 Messaging API

A clean, simple API is provided to send and receive messages in RapidSMS. For most cases, you'll just need to use the `send` and `receive` functions outlined below.

8.1.1 Receiving Messages

`rapidsms.router.`**`receive`**(*text*, *connection*, ***kwargs*)

Creates an incoming message and passes it to the router for processing.

Parameters

- **`text`** – text message
- **`connection`** – RapidSMS *`Connection`* object
- **`kwargs`** – Extra kwargs to pass to *`IncomingMessage`* constructor

Returns *`IncomingMessage`* object constructed by router. A returned message object does not indicate that router processing has finished or even started, as this depends on the router defined in *`RAPIDSMS_ROUTER`*.

Return type *`IncomingMessage`*

To receive a message, you can use the `receive` function, which will automatically create an *`IncomingMessage`* and pass it to your router to be processed. Typically, backends will make the most use of `receive`, but it can be used anywhere within your application to route an incoming message (such as from a Django view).

Here's an example using `receive`:

```
from rapidsms.router import receive
receive("echo hello", connection)
```

This sends a message to the router saying `echo hello` was received from a *Connection* object. You can find more examples of `receive` in the official RapidSMS backends.

class rapidsms.messages.incoming.**IncomingMessage**(**args*, ***kwargs*)
Bases: *rapidsms.messages.base.MessageBase*

Inbound message that provides an API to handle responses.

respond(*text*, ***kwargs*)
Respond to this message, sending the given text to the connections that this message came from.

Responses are saved, and sent after incoming processing phases are complete.

Arbitrary arguments are passed along to the *send()* method.

Parameters

- **text** (*string*) – The text of the message
- **connections** (list of *Connection*) – (optional) send to a different set of connections than were in the incoming message.
- **in_response_to** (*MessageBase*) – (optional) the message being responded to.

Returns dictionary with the arguments that will be passed to *rapidsms.router.send()* to send this response.

responses = None
list of messages created by IncomingMessage.respond()

8.1.2 Sending Messages

rapidsms.router.**send**(*text*, *connections*, ***kwargs*)
Creates an outgoing message and passes it to the router to be processed and sent via the respective backend.

Arbitrary arguments are passed along to *new_outgoing_message()*.

Parameters

- **text** – text message
- **connections** – list or QuerySet of RapidSMS *Connection* objects
- **kwargs** – Extra kwargs to pass to *OutgoingMessage* constructor

Returns message constructed by router. A returned message object does not indicate that router processing has finished or even started, as this depends on the router defined in *RAPIDSMS_ROUTER*.

Return type *OutgoingMessage*

It's just as easy to send a message using RapidSMS. You can send a message from anywhere within your application. Here's an example using `send`:

```
from rapidsms.router import send
send("hello", connections)
```

This sends `hello` to each identity and backend associated with the `connections` object. You can find more examples of `send` in the official RapidSMS backends.

class rapidsms.messages.outgoing.**OutgoingMessage**(**args*, ***kwargs*)
Bases: *rapidsms.messages.base.MessageBase*

Outbound message that can easily be sent to the router.

extra_backend_context()
Specific metadata to be included when passed to backends.

send()
Send the message. Equivalent to `rapidsms.router.send(text, connections)`.

8.1.3 MessageBase

Both incoming and outgoing message classes inherit from a common class, MessageBase.

class rapidsms.messages.base.**MessageBase**(*connections=None*, *text=None*, *id_=None*, *in_response_to=None*, *fields=None*, *connection=None*)
Basic message representation with text and connection(s).

connection
The first *Connection* - *deprecated*.

connections = None
The connections this message was received from or sent to. A list of *Connection*

contact
The first connection's *Contact* - *deprecated*

fields = None
fields can be used to pass along arbitrary metadata

static generate_id()
Create a random unique ID for this message object.

handled = None
a message can be marked "handled" by any app, which will short-circuit the default phase in the router.

id = None
a unique ID for this message

in_response_to = None
link back to original message if this is a response

peer
Return the identity (eg. a phone number, email address, irc nickname) on the other end of this message. But you shouldn't use this method. It only seems to encourage people to ignore the distinction between backends and identities, and create fields like "mobile_number", which is all kinds of wrong. *deprecated*

processed = None
a message is considered "unprocessed" until rapidsms has dispatched it to all apps.

raw_text = None
save original text for future reference

text = None
the message

8.1.4 ErrorMessage

There's also an ErrorMessage class that can be used when sending error messages.

class rapidsms.messages.error.**ErrorMessage**(**args*, ***kwargs*)
Bases: *rapidsms.messages.outgoing.OutgoingMessage*

8.1.5 Contacts

RapidSMS represents entities that it can communicate with using a *Contact* object. Each Contact has a name. A Contact could represent a person, an organization, or any other entity that might have a phone, email account, etc.

class rapidsms.models.**Contact**(**args*, ***kwargs*)
Bases: *rapidsms.models.ContactBase*

This model represents a person with a name

Most of a Contact is represented in the ContactBase class:

class rapidsms.models.**ContactBase**(**args*, ***kwargs*)

created_on = None
when the contact was created

default_connection
Return the default connection for this person. Currently this arbitrarily returns the first connection.

is_anonymous
Return True if the individual has no name

language = None
The contact's preferred language. the spec: http://www.w3.org/International/articles/language-tags/Overview reference: http://www.iana.org/assignments/language-subtag-registry

modified_on = None
when the contact was last modified

name = None
The individual's name (optional)

8.1.6 Connections

RapidSMS might be able to communicate with an entity represented by a Contact in multiple ways. The entity could have several phone numbers, email addresses, etc.

RapidSMS uses a *Connection* to represent each way of communicating with a Contact. Each Connection specifies a backend to use, and how the entity is identified on that backend. The identifier is called an `identity`, and depending on the backend, it could be a phone number, email address, or something else. Most RapidSMS code should not make any assumptions about the format of identities.

class rapidsms.models.**Connection**(**args*, ***kwargs*)
Bases: *rapidsms.models.ConnectionBase*

This model pairs a Backend object with an identity unique to it (eg. a phone number, email address, or IRC nick), so RapidSMS developers need not worry about which backend a messge originated from.

Most of a Connection is represented in the ConnectionBase class:

class rapidsms.models.**ConnectionBase**(**args*, ***kwargs*)

backend
foreign key to this connection's *BackendBase*

contact
(optional) associated *Contact*

created_on = None
when this connection was created

identity = None
unique identifier for this connection on this backend (e.g. phone number, email address, IRC nick, etc.)

modified_on = None
when this connection was last modified

8.1.7 Connection Lookup

rapidsms.router.**lookup_connections**(*backend*, *identities*)
Find connections associated with backend and identities. A new connection object will be created for every backend/identity pair not found.

Parameters

- **backend** – backend name (as a string) or *BackendBase* object
- **identities** – list of identities to find associated with the backend

Returns List of *Connection* objects

Since most of the time you'll need to find connections for a backend and phone number, RapidSMS has a helper function, `lookup_connections`, to do the lookup for you. Additionally, if the backend and phone number connection doesn't exist, it'll be created automatically. Here's an example of `lookup_connections`:

```
from rapidsms.router import send, lookup_connections
connections = lookup_connections(backend="example-backend",
                                 identities=['1112223333'])
send("hello", connections=connections)
```

8.2 BlockingRouter

Please, see the release notes The *BlockingRouter* is the most basic and easy to use router included with RapidSMS. For this reason it is also the default router. As its name suggests, *BlockingRouter* handles messages synchronously (within the main HTTP thread), waiting for application and backend processing to complete before continuing. This is acceptable for many scenarios, but will be less efficient if your project needs to handle many inbound and outbound messages.

8.2.1 Installation

Set *RAPIDSMS_ROUTER* to use *BlockingRouter*:

```
RAPIDSMS_ROUTER = "rapidsms.router.blocking.BlockingRouter"
```

That's it!

8.2.2 How it works

By default, *BlockingRouter* automatically adds apps and backends defined in your settings file via INSTALLED_APPS and *INSTALLED_BACKENDS*. If you instantiate a *BlockingRouter*, you can see the available apps and backends:

```
>>> from rapidsms.router.blocking import BlockingRouter
>>> router = BlockingRouter()
>>> router.apps
[<app: handlers>, <app: default>, <app: locations>, <app: messagelog>]
>>> router.backends
{'message_tester': <backend: message_tester>}
```

In this scenario, these settings were used:

```
INSTALLED_APPS = [
    # trimmed to only show the relevant apps
    "rapidsms.contrib.handlers",
    "rapidsms.contrib.default",
    "rapidsms.contrib.locations",
    "rapidsms.contrib.messagelog",
]

INSTALLED_BACKENDS = {
    "message_tester": {
        "ENGINE": "rapidsms.contrib.httptester.backend.HttpTesterCacheBackend",
    },
}
```

8.2.3 Implementation

BlockingRouter is the default router, but it is also the base router for all RapidSMS routers. *CeleryRouter* and *DatabaseRouter* extend *BlockingRouter* and override necessary functionality. A subset of its methods are outlined below:

class rapidsms.router.blocking.**BlockingRouter**(**args*, ***kwargs*)

Base RapidSMS router implementation.

add_app(*module_name*)

Add RapidSMS app to router. Installed apps will be notified of incoming and outgoing messages. If module_name is a Django app, the method looks in app_name.app for an AppBase subclass to use.

Parameters **module_name** – AppBase object or dotted path to RapidSMS app.

Returns AppBase object if found, otherwise None.

add_backend(*name*, *module_name*, *config=None*)

Add RapidSMS backend to router. Installed backends will be used to send outgoing messages.

Parameters

- **name** – Name of backend.
- **module_name** – *BackendBase* object or dotted path to backend class.

Returns *BackendBase* object if found, otherwise a ValueError exception will be raised.

get_app(*module_name*)

Access installed app by name.

Parameters **module_name** – Dotted path to RapidSMS app.

Returns `AppBase` object if found, otherwise `None`.

incoming_phases = ('filter', 'parse', 'handle', 'default', 'cleanup')
Incoming router phases processed in the order in which they're defined.

new_incoming_message(*text*, *connections*, *class_=<class 'rapidsms.messages.incoming.IncomingMessage'>*, ***kwargs*)
Create and return a new incoming message. Called by *send*. Overridable by child-routers.

Parameters

- **text** – Message text
- **connections** – List or QuerySet of `Connection` objects
- **class** – Message class to instaniate

Returns *IncomingMessage* object.

new_outgoing_message(*text*, *connections*, *class_=<class 'rapidsms.messages.outgoing.OutgoingMessage'>*, ***kwargs*)
Create and return a new outgoing message. Called by *receive*. Overridable by child-routers.

Parameters

- **text** – Message text
- **connections** – List or QuerySet of `Connection` objects
- **class** – Message class to instaniate

Returns *OutgoingMessage* object.

outgoing_phases = ('outgoing',)
Outgoing router phases processed in the order in which they're defined.

receive_incoming(*msg*)
All inbound messages will be routed through `receive_incoming` by *send*. `receive_incoming` is typically overridden in child routers to customize incoming message handling.

Parameters **msg** – *IncomingMessage* object

send_outgoing(*msg*)
All outbound messages will be routed through `send_outgoing` by *receive*. `send_outgoing` is typically overridden in child routers to customize outgoing message handling.

Parameters **msg** – *OutgoingMessage* object

class `rapidsms.router.blocking.router.`**BlockingRouter**
is the full name for *rapidsms.router.blocking.BlockingRouter*.

8.3 CeleryRouter

Please, see the release notes *CeleryRouter* uses Celery to queue incoming and outgoing messages.

BlockingRouter processes messages synchronously in the main HTTP thread. This is fine for most scenarios, but in some cases you may wish to process messages outside of the HTTP request/response cycle to be more efficient. *CeleryRouter* is a custom router that allows you to queue messages for background processing. It's designed for projects that require high message volumes and greater concurrency.

8.3.1 Installation

Note: *CeleryRouter* depends on django-celery 3.0+. Please follow the setup instructions in Scheduling Tasks with Celery before proceeding.

Add `rapidsms.router.celery` to `INSTALLED_APPS`, then import djcelery and invoke `setup_loader()`:

```
INSTALLED_APPS = (
    # Other apps here
    "rapidsms.router.celery"
)
import djcelery
djcelery.setup_loader()
```

This will register Celery tasks in `rapidsms.router.celery.tasks`.

Set *RAPIDSMS_ROUTER* to use *CeleryRouter*:

```
RAPIDSMS_ROUTER = "rapidsms.router.celery.CeleryRouter"
```

That's it!

8.3.2 Celery workers

Finally, you'll need to run the celery worker command (in a separate shell from `runserver`) to begin consuming queued tasks:

```
python manage.py celery worker -lDEBUG
```

Now your messages will be handled asynchronously with *CeleryRouter*.

8.3.3 Configuration

Eager backends

Sometimes your project may require the use of a synchronous backend. If this is the case, you can configure specific backends to utilize Celery's eager functionality with the `router.celery.eager` backend setting. For example, here's how you can force the httptester backend to be eager:

```
INSTALLED_BACKENDS = {
    "message_tester": {
        "ENGINE": "rapidsms.contrib.httptester.backend",
        "router.celery.eager": True,
    },
}
```

Using this setting means that the task will be executed in the current process, and not by an asynchronous worker. Please see the Celery documentation for more information on calling tasks.

Logging

Note: Please see the Django logging documentation for further information regarding general logging configuration.

All logging specific to *CeleryRouter* is handled through the `rapidsms.router.celery` name. For example, if you have a `file` handler defined, you can capture all messages using the following configuration:

```
LOGGING_CONFIG = {
    'rapidsms.router.celery': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

8.4 DatabaseRouter

Please, see the release notes *DatabaseRouter* provides the following functionality:

- All inbound and outbound messages are stored in the database.
- Inbound and outbound messages are processed asynchronously with Celery.
- Outbound messages are automatically split into batches for sending.
- Use of Django's bulk create to optimize database inserts.
- Messages that fail to send will use Celery's retry mechanism.

Similar to *CeleryRouter*, *DatabaseRouter* is designed for projects that require high messages volumes.

8.4.1 How it works

- Before doing any processing, an inbound message is loaded into the `Message` and `Transmission` models. A celery task is then queued to process the message asynchronously.
- The celery task reconstructs the message object, fires up the router, and passes it off for inbound processing.
- Any replies are loaded into the `Message` and `Transmission` models.
- The router then divides the outbound messages by backend and queues tasks for sending chunks of messages to their respective backends.
- As tasks complete, the status of the messages are updated in the database, including any errors that occurred.

8.4.2 Installation

Note: *DatabaseRouter* depends on django-celery 3.0+. Please follow the django-celery setup instructions before proceeding.

Add `rapidsms.router.db` to `INSTALLED_APPS`, then import djcelery and invoke `setup_loader()`:

```
INSTALLED_APPS = (
    # Other apps here
    "rapidsms.router.db",
)
import djcelery
djcelery.setup_loader()
```

This will register Celery tasks in `rapidsms.router.db.tasks`.

Set *RAPIDSMS_ROUTER* to use *DatabaseRouter*:

```
RAPIDSMS_ROUTER = "rapidsms.router.db.DatabaseRouter"
```

Run `syncdb` to create the necessary database tables:

```
python manage.py syncdb
```

That's it!

8.4.3 Configuration

The database router has one optional setting, *DB_ROUTER_DEFAULT_BATCH_SIZE*, to change the default maximum size of a batch of messages from 200.

Celery workers

Finally, you'll need to run the celery worker command (in a separate shell from `runserver`) to begin consuming queued tasks:

```
python manage.py celery worker --loglevel=info
```

Now your messages will be handled asynchronously with *DatabaseRouter*.

8.4.4 Database models

DatabaseRouter utilizes two database models, `Message` and `Transmission`.

Message

The `Message` model contains the context of a text message. For every associated `Connection`, a `Message` has an associated `Transmission`.

Transmission

A `Transmission` represents the instance of a particular `Message` and `Connection`.

Message status values

`Message` and `Transmission` objects can have the following status values:

- **Inbound values:**
  - `Q` - *Queued*: Message is queued and awaiting processing
  - `R` - *Received*: Message has been processed and responses are queued
  - `E` - *Errored*: An error occurred during processing
- **Outbound values:**
  - `Q` - *Queued*: Message is queued and awaiting processing
  - `P` - *Processing*: Message is sending
  - `S` - *Sent*: All associated transmissions have been sent

- `D` - *Delivered*: All associated transmissions have been delivered (requires backend functionality)
- `E` - *Errored*: An error occurred during processing

8.5 Choosing a Router

Each RapidSMS project can use only one router, which should be chosen based on the needs of the project. The path to your chosen router must go in the *RAPIDSMS_ROUTER* setting:

```
RAPIDSMS_ROUTER = 'path.to.your.RouterClass'
```

The default router is `rapidsms.router.blocking.BlockingRouter`.

The choice of router is an important decision that will affect your message processing performance. For example, some routers are easy to set up but will struggle with large message volumes. More complex routers may process messages more efficiently, but require more work to set up.

8.5.1 Supplied Routers

RapidSMS includes several routers for you to use:

- *BlockingRouter* - Default router that processes messages synchronously within the HTTP thread.
- *CeleryRouter* - Celery-enabled router that processes messages asynchronously.
- *DatabaseRouter* - Database, Celery-enabled router that queues messages in the database for asynchronous processing.

Here are some characteristics of the supplied routers. *B* is the blocking router, *C* is the Celery router, and *D* is the database router.

B	C	D	Characteristic
n	n	y	Stores messages in database
n	y	y	Requires Celery
y	n	n	Delays in one operation can block all other operations
n	n	y	Can recover and retry failed sends
n	n	y	Keeps a record of which messages have been sent and whether the send was successful

If you can't find a router that's suitable for your needs, you can write a custom router.

8.5.2 Using a custom router

While RapidSMS includes support for a number of routers out-of-the-box, sometimes you may want to use a customized router. To use a custom router with RapidSMS, use the dotted Python path to the router class for the *RAPIDSMS_ROUTER* setting, like so:

```
RAPIDSMS_ROUTER = 'path.to.RouterClass'
```

If you're building your own router, you can use the standard routers as reference implementations. All routers should extend from *BlockingRouter*.

8.6 Applications and Backends

While the router provides the foundation for messaging processing, applications and backends actually perform the message processing:

- **Applications:** The router maintains a collection of related applications through which it routes incoming and outgoing messages. Applications are defined in `INSTALLED_APPS` and loaded, by default, when the router is instantiated via *`add_app`*.
- **Backends:** The router also maintains a collection of related backends to send outgoing messages. Backends are defined in *`INSTALLED_BACKENDS`* and loaded, by default, when the router is instantiated via *`add_backend`*.

8.7 Message Processing

The Messaging API defines *`send`* and *`receive`* to route messages through the router. Messages are processed via a series of phases, depending on direction. These phases are outlined below.

8.7.1 Incoming Messages

Note: See also the *application documentation on incoming message processing*.

Incoming messages are processed in five phases. Each application provides code for executing the phases. The router method defines hooks which allow an application to filter out a message, skip phases, or stop further processing.

1. *filter* - **Optionally abort all further processing of the incoming message (including cleanup).**
2. *parse* - **Modify the message in a way that is globally useful.**
3. *handle* - **Respond to the incoming message.**
4. *default* - **Execute a default action if no application returns true during the handle phase.**
5. *cleanup* - **Clean up work from previous phases.**

The order in which the router chooses applications to process messages is extremely important, because each application will have the opportunity to block subsequent applications from processing a message. *`receive_incoming`* processes messages in the order they are listed in `INSTALLED_APPS`.

8.7.2 Outgoing Messages

Note: See also the *application documentation on outgoing message processing*.

`send_outgoing` processes messages sequentially, in the order they are listed in `INSTALLED_APPS`. However, the applications are called in reverse order, so the first application called to process an incoming message is the last application that is called to process an outgoing message. If any application returns `False` during the *outgoing* phase, all further processing of the message will be aborted.

CHAPTER 9

Using virtualenv

We highly recommend using virtualenv and virtualenvwrapper to work on RapidSMS. These tools provide isolated Python environments, which are more practical than installing packages system wide. They also allow installing packages without administrator privileges.

1. **Install virtualenv and virtualenvwrapper.** Use pip to install the latest version (and upgrade if you have an older copy):

```
sudo pip install --upgrade virtualenv
sudo pip install --upgrade virtualenvwrapper
```

Then follow the virtualenvwrapper install docs to setup your shell properly.

2. **Create a new virtual environment for RapidSMS.** Now we'll create a new virtual environment to isolate our development:

```
mkvirtualenv --distribute --no-site-packages rapidsms
```

3. **Remember to activate your virtualenv.** If you restart or need to return to your virtualenv at any point, you can easily reactivate it:

```
workon rapidsms
```

CHAPTER 10

Settings

Here is a full list of available settings, and their default values, for RapidSMS and its contrib apps.

10.1 DB_ROUTER_DEFAULT_BATCH_SIZE

App rapidsms.router.db

Default 200

The default maximum batch size when the database router is sending messages in bulk.

10.2 DEFAULT_RESPONSE

App rapidsms.contrib.default

Default `'Sorry, %(project_name)s could not understand your message.'`

The default response to an *IncomingMessage* that is not handled by any other application. To include *PROJECT_NAME*, use `'%(project_name)s'` in the string.

10.3 EXCLUDED_HANDLERS

App rapidsms.contrib.handlers

Default `[]`

Deprecated since version 0.15.0: See *RAPIDSMS_HANDLERS* instead.

The rapidsms.contrib.handlers application will not load any handler in a module that is in this list. The module name of each handler is compared to the value in this list using prefix matching. For more information, see *Handler Discovery*.

10.4 INSTALLED_BACKENDS

Default *not set*

This setting is a dictionary that allows you to configure backends for use in your project. There is no default value as the needs of individual projects vary widely. This setting mimics the structure of the Django DATABASES setting, with the following general format:

```
INSTALLED_BACKENDS = {
    'backend1_name': {
        'ENGINE': 'path.to.backend1.BackendClass',
    },
    'backend2_name': {
        'ENGINE': 'path.to.backend2.BackendClass',
    },
}
```

Each backend dictionary requires only the 'ENGINE' key, which defines the Python path to the backend. Other key-value pairs can be used to configure backend-specific options.

Example configuration:

```
INSTALLED_BACKENDS = {
    "message_tester": {
        "ENGINE": "rapidsms.contrib.httptester.backend.HttpTesterCacheBackend",
    },
}
```

10.5 INSTALLED_HANDLERS

App rapidsms.contrib.handlers

Default `None`

Deprecated since version 0.15.0: See *RAPIDSMS_HANDLERS* instead.

If this setting is not `None`, the rapidsms.contrib.handlers application will only load handlers in modules that are in this list. The module name of each handler is compared to each value in this list using prefix matching. For more information see *Handler Discovery*.

10.6 PROJECT_NAME

Default `'RapidSMS'`

The name of your project. This is used by some applications such as rapidsms.contrib.default to customize message responses.

10.7 RAPIDSMS_HANDLERS

Please, see the release notes

App rapidsms.contrib.handlers

Default `[]`

A list of names of the handler classes that should be loaded. For more information see *Handler Discovery*.

If this is set, it overrides the older, deprecated behavior of loading all handlers, modified by *INSTALLED_HANDLERS*, *EXCLUDED_HANDLERS*, and *RAPIDSMS_HANDLERS_EXCLUDE_APPS*.

10.8 RAPIDSMS_HANDLERS_EXCLUDE_APPS

App rapidsms.contrib.handlers

Default `[]`

Deprecated since version 0.15.0: See *RAPIDSMS_HANDLERS* instead.

The rapidsms.contrib.handlers application will not load handlers from any Django app included in this list. For more information see *Handler Discovery*.

10.9 RAPIDSMS_ROUTER

Please, see the release notes

Default `'rapidsms.router.blocking.BlockingRouter'`

The router is used to handle incoming and outgoing messages. For the list of available routers see RapidSMS Routers.

CHAPTER 11

Translation

RapidSMS uses Django's internationalization (i18n) architecture to allow you to make your project or application translatable. This documentation assumes you're already familiar with this architecture. If you're not, we recommend reviewing Django's translation documentation before proceeding.

This page will start with a brief review of Django translation, then move on to RapidSMS translation.

11.1 Introduction

You can specify a translation string by using the function `ugettext`. The convention is to import this as a shorter alias, _, to save typing. In this example, the text `"Welcome to RapidSMS."` is marked as a translation string:

```
from django.utils.translation import ugettext as _

def do_something():
    output = _("Welcome to RapidSMS.")
```

By default, Django will attempt to translate this string based on your `LANGUAGE_CODE` setting.

RapidSMS will not automatically attempt to translate routed messages. This is an intential decision to require application developers to explicity initiate the message translation process. You can use the methods below to translate messages.

11.2 Language specification

To set a specific language, you can use Django's `translation.override` context manager:

```
from django.utils import translation
from django.utils.translation import ugettext as _

def do_something():
    with translation.override("es"):  # Spanish
        message1 = _("Welcome to RapidSMS.")
    with translation.override("fr"):  # French
        message2 = _("Welcome to RapidSMS.")
```

11.3 Contact language setting

Every RapidSMS `Contact` has an associated `language` field. This field is used to specify the contact's language preference. You can use this field to send messages in the correct language.

11.3.1 Message responses

It's quite common to use `msg.respond()` within your application to respond to messages. To send a response in the contact's preferred language, you can use `translation.override` with their language:

```
from django.utils import translation
from django.utils.translation import ugettext as _
from rapidsms.apps.base import AppBase

class HelloApp(AppBase):

    def handle(self, msg):
        if msg.text == "hello":
            with translation.override(msg.connection.contact.language):
                msg.respond(_('hello'))
            return True
```

This example will attempt respond with `"hello"` in the contact's preferred language.

11.3.2 Sending messages

To ease translating messages to multiple connections, RapidSMS provides a utility function, `group_connections`, to divide connections into groups by their contact's language setting.

This example illustrates how you could attempt to say *hello* in the correct language(s) without needing to inspect each connection manually:

```
from django.utils import translation
from django.utils.translation import ugettext as _
from rapidsms.utils import translation as trans_helpers
from rapidsms.router import send

def say_hello_to_everyone():
    connections = Connection.objects.all()
    for lang, conns in trans_helpers.group_connections(connections):
        with translation.override(lang):
            send(_('hello'), conns)
```

CHAPTER 12

Intro to Extensible Models

Note: Extensible models will be removed in a future version of RapidSMS. We do not recommend using them in any new code.

This is a brief summary of some sandbox work to better understand Extensible Models. This is "not" a page on how to make extensible models, the Contact model in the rapid core is a fine example of that. This is intended to be a detailed example of how to set up your folder structure and stay sane while extending an Extensible Model.

12.1 Initial Setup

For this sandbox we'll be extending the Contact model in three separate apps, to see what exactly happens at the DB level. I started with a clean install of rapid, and then created three apps with creative names:

```
$ python manage.py startapp testextensions_main
$ python manage.py startapp testextensions_clash
```

The folder structure for each of these is the same:

```
testextensions_main/models.py
testextensions_main/extensions
testextensions_main/extensions/rapidsms
testextensions_main/extensions/rapidsms/contact.py
testextensions_main/extensions/rapidsms/__init__.py
testextensions_main/extensions/__init__.py
testextensions_main/tests.py
testextensions_main/views.py
testextensions_main/__init__.py
```

These are all fairly dumb apps for the sake of this example, so only contact.py has any pertinent content:

```
from django.db import models

class TestContact(models.Model):
    is_used = models.BooleanField(default=True)

    class Meta:
        abstract = True
```

The folder structure here is very important: we're extending the Contact model, within the rapidsms app. So the modeule where our extension class exists "must" be myapp/extensions/rapidsms/contact.py. The name of the class

itself, TestContact, is unimportant, however it "must" be abstract. Any abstract models within this module will have their attributes added to the base Contact class.

In testextensions_clash, I used the same exact TestContact model (it also has a boolean is_used attribute).

Initially, I started with a clean install of rapidsms, without any of these sample apps in the INSTALLED_APPS under settings. I did this to demonstrate how one might add a new app to an already-running instance, pulling in new extensions to an existing model.

12.2 Extension Experiments

Firstly, installing south and placing all the Extensible Models under version control makes it easy to automatically add columns as you pull in new apps that extend them (if you aren't using it already).

```
$ easy_install south
```

You will likely want to change the place where the 'migrations' folder exists within your project (http://south.aeracode.org/docs/settings.html#setting-south-migration-modules), otherwise it'll place them directly with in rapidsms/lib (probably not a Good Idea). In my settings.py, I added:

```
SOUTH_MIGRATION_MODULES = {
    'rapidsms': 'testextensions_main.migrations',
}
```

Now we place rapidsms under migration control, but this command doesn't create any tables.

```
$ python manage.py schemamigration rapidsms --initial
 + Added model rapidsms.Backend
 + Added model rapidsms.App
 + Added model rapidsms.Contact
 + Added model rapidsms.Connection
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate rapidsms
```

This creates all tables except the core rapidsms tables:

```
$ python manage.py syncdb
Creating table ...
...
Synced:
 > south
 > rapidsms.contrib.handlers
 > django.contrib.sites
 > django.contrib.auth
 > django.contrib.admin
 > django.contrib.sessions
 > django.contrib.contenttypes
 > rapidsms.contrib.locations
 > rapidsms.contrib.messagelog
 > testextensions_clash

Not synced (use migrations):
 - rapidsms
```

The following creates the rapidsms tables (migration-controlled):

```
$ python manage.py migrate
 - Migrating forwards to 0001_initial.
 > rapidsms:0001_initial
```

We'll now have a rapidsms_contact table with the following structure:

```
CREATE TABLE "rapidsms_contact" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(100) NOT NULL,
    "language" varchar(6) NOT NULL
);
```

Now we can demonstrate a few things, the first of which is how to pull in a new app with extensions and automatically update the contact db. At this point, I added my app, testextensions_main to the INSTALLED_APPS in settings.py:

```
$ python manage.py schemamigration rapidsms --auto

 + Added field is_used on rapidsms.Contact
 Created 0002_auto__add_field_contact_is_used.py. You can now apply this migration with: ./manage.py

$ python manage.py migrate rapidsms

 - Migrating forwards to 0002_auto__add_field_contact_is_used.
 > rapidsms:0002_auto__add_field_contact_is_used
```

Steps 6 and 7 auto-magically added my additional column to the contacts table!

```
CREATE TABLE "rapidsms_contact" (
    "is_used" bool NOT NULL DEFAULT True,
    "id" integer PRIMARY KEY,
    "language" varchar(6),
    "name" varchar(100));
```

For anyone more knowledgeable of the way ExtensibleBase works, this may not be as big a deal, but for me the implications were pretty exciting...provided that one keeps the extensible models under migration control, you can add new apps after your initial deployment, extending these models with more and more columns as you go...

As a final demonstration, just to show one (unsurprising) limitation of extensible models is that two apps cannot extend the same model with a column of the same name. Let's add testextensions_clash to the INSTALLED_APPS to see what happens:

```
$ python manage.py schemamigration rapidsms --auto

Nothing seems to have changed.
```

Hmmm...interesting! We have two extensions that are both wanting to add the same column, and south sees them as having no problems. It merges these two concepts together (which could be desired or a really Bad Thing, depending on what you're wanting).

Blow away the database, remove south support, and just trying syncing the db the regular way, with both _main and _clash apps installed:

```
  $ python manage.py syncdb
  Syncing...
  Creating table south_migrationhistory
  Creating table rapidsms_backend
  Creating table rapidsms_app
  Creating table rapidsms_contact
  Traceback (most recent call last):
   ......
File "/home/david/Projects/CoreDevRapid/env/lib/python2.6/site-packages/Django-1.2.1-py2.6.egg/django
  return Database.Cursor.execute(self, query, params)
  django.db.utils.DatabaseError: duplicate column name: is_used
```

In this case the clash is identified and in fact impossible to create.

12.3 Conclusions

South provides an easy way to add new attributes to ExtensibleModels, within an already-deployed instance of RapidSMS.

Depending on your needs, south-managed migrations and regular syncdb offer different behaviors for attribute clashes with extensible models used by two separate apps. In either case, if two groups within the community are working on apps that extend the same model (and that both use one another's apps), they should probably be coordinating regularly when adding attributes, to be sure there are no clashes, and to determine which attributes should be brought into the base class.

CHAPTER 13

Front End

13.1 Introduction

RapidSMS provides an optional framework for making the applications on your site have a common look and feel.

Starting with release 0.12.0, the framework uses Twitter Bootstrap, v2.2.2.

This document is intended to describe how the contrib apps use Bootstrap, and how you might use it in your own apps.

13.2 Base template

Your templates should extend `base.html`, either directly or indirectly. That will load Bootstrap and set up the common page structure for RapidSMS apps.

The simplest way to provide content for your page is to override `{% block content %}`. That block is rendered inside of a Bootstrap fluid row like this:

```
<div class="row-fluid content">
   {% block content %}{% endblock %}
</div>
```

You can then divide your page into columns using Bootstrap `span` classes:

```
{% block content %}
  <div class="span6">  Left side   </div>
  <div class="span6">  Right side  </div>
{% endblock %}
```

See the Bootstrap documentation for more things you can do inside a fluid row to affect the layout.

13.3 Title

Set the page title with `{% block title %}`:

```
{% block title %}Send a Text Message{% endblock title %}
```

13.4 Additional styles

If you have a page that needs additional stylesheets, you can override `{% block extra_stylesheets %}`:

```
{% block extra_stylesheets %}
  <link type="text/css" rel="stylesheet"
        href="{{ STATIC_URL }}my-app/stylesheets/my-app.css" />
{% endblock %}
```

13.5 Additional javascript

Additional javascript can be added by overriding `{% extra_javascript %}`:

```
{% block extra_javascript %}
  <script src="{{ STATIC_URL }}my-app/js/my-app.js"
          type="text/javascript"></script>
{% endblock %}
```

13.6 Page header

To display a header at the top of the page in the same style as other apps, use the `page-header` class and `<h1>`. If you need to divide the page into columns after that, you can include a div and then put your `span` divs inside that to keep everything organized:

```
{% block content %}
  <div class="page-header">
    <h1>My Application Page Header</h1>
  </div>
  <div>
    <div class="span6">  Left side   </div>
    <div class="span6">  Right side  </div>
  </div>
{% endblock %}
```

13.7 Top menu

The menu at the top of the page is a Bootstrap navigation bar.

Your site can configure the links that appear between the RapidSMS logo and the login/logout link by overriding the `rapidsms/_nav_bar.html` template. You can do this by creating a file with this path from the top of your project: `templates/rapidsms/_nav_bar.html`.

Your template will be included in the base page template. It should contain `<li>` elements for each link. Example:

```
{% load url from future %}
<li><a href="{% url 'app1' %}">App1</a></li>
<li><a href="{% url 'app2' %}">App2</a></li>
<li><a href="{% url 'app3' %}">App3</a></li>
```

Note: Keep these links short. If the links take up too much room on the page, they will wrap in the header, forcing the bottom of the page header down and overlapping part of the page.

13.7.1 Login/Logout links

The login or logout link can be removed or replaced by overriding the *login_link* block. Here's the default value:

```
{% block login_link %}
  <li>
    {% block auth %}
      {% if user.is_authenticated %}
        <a href="{% url 'rapidsms-logout' %}">
          {% trans "Log out" %} {{ user.username }}
        </a>
      {% else %}
        <a href="{% url 'rapidsms-login' %}">{% trans "Log in" %}</a>
      {% endif %}
    {% endblock auth %}
  </li>
{% endblock %}
```

13.7.2 Admin link

Similarly, a link to the Django admin is shown for staff users who are logged in. Change that by overriding the *admin_link* block. Here's the default value:

```
{% block admin_link %}
  {% if user.is_staff %}
    <li>
      <a href="{% url 'admin:index' %}">{% trans "Admin" %}</a>
    </li>
  {% endif %}
{% endblock %}
```

13.8 Tables

To include tables in a page, the django_tables2 package works well. Look at the *rapidsms.contrib.messagelog* app for an example. Note particularly how the view overrides the default template used by *django_tables2* to use one that takes advantage of Bootstrap styling.

13.9 Forms

Bootstrap can make forms look nice too. RapidSMS's form tags have been updated to work well with Bootstrap. The `render_form` tag will render your form's data fields correctly for Bootstrap. Then all you have to do is add any submit buttons you need, properly marked up. See the Bootstrap documentation for full details, but here's an example from another contrib app, rapidsms.contrib.httptester:

```
{% extends "base.html" %}
{% load forms_tags %}
...
{% block content %}
  <div class="page-header">
    <h1>Message Tester</h1>
  </div>
```

```
  <div>
    <div class="span4">
      <div>
        <form action="" method="post" enctype="multipart/form-data">
          {% render_form message_form %}
          {% csrf_token %}

          <div class="form-actions">
            <button type="submit" class="btn btn-primary" id="send-btn" name="send-btn">Send</buttor
            <label for="send-btn">Send single or multiple messages</label>
            ...
          </div>
        </form>
      </div>
    </div>
    <div class="span8">
    ...
     </div>

  </div>
{% endblock %}
```

13.10 Messages to Users

Please, see the release notes If you use the Django messages framework to send messages to your users, the base template will display them nicely above the page content.

CHAPTER 14

Logging

It's good practice to log a message using Python logging whenever an error or exception occurs. There are a myriad of tools administrators can then use to send the information where they want it, send email alerts, analyze trends, etc.

If you want to log in your app, just:

```
import logging
logger = logging.getLogger(__name__)
```

and use:

```
logger.debug("msg")
logger.critical("msg")
logger.exception("msg")
# etc.
```

All RapidSMS core logging can now be captured using the `'rapidsms'` root logger. (There's not a lot of logging from the core yet, but pull requests are welcome.)

For example, if you wanted messages from the RapidSMS core to be written to a file *"/path/rapidsms.log"*, you could define a new handler in the `LOGGING` setting in Django:

```
LOGGING = {
    ...
    'handlers': {
        ...
        'rapidsms_file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': '/path/rapidsms.log',
        },
        ...
    },
    ...
}
```

Setting `level` to `DEBUG` means all messages of level DEBUG and lower will be passed through (that's all of them). Then this handler will write those messages to the file `/path/rapidsms.log`. They'll be formatted by the default formatter.

Then configure the `rapidsms` logger to send messages to that handler:

```
LOGGING = {
    ...
    'loggers': {
        'rapidsms': {
```

```
            'handlers': ['rapidsms_file'],
            'propagate': True,
            'level': 'DEBUG',
        },
    },
    ...
}
```

Setting `level` to `DEBUG` means all messages of level DEBUG and lower will be passed through (that's all of them).

The logger name `rapidsms` means any logger to a name that matches that (`rapidsms`, `rapidsms.models`, etc) will be passed to this handler to handle.

Setting `propagate` to `True` means the same messages will be passed to other handlers if they also match. (This handler does not consume the messages.)

If you created your project with the latest *Rapidsms project template* and haven't changed the settings, all rapidsms logging will be written to *rapidsms.log* in your project directory.

CHAPTER 15

Testing RapidSMS Applications

Automated testing is an extremely useful tool and, therefore, we recommend writing tests for all RapidSMS applications and projects. Tests provide a way to repeatedly ensure that your code functions as expected and that new code doesn't break existing functionality.

This document outlines the tools and best practices for writing RapidSMS tests.

15.1 Prerequisites

A RapidSMS test is written using standard Python and Django testing utilities. If you're unfamiliar with these concepts, please take a moment to read through the following links:

- Python's unittest module
- **Django's Testing documentation, including:**
  - Writing tests
  - Running tests
  - Testing tools

Additionally, since much of RapidSMS is Django-powered, these docs will not cover testing standard Django aspects (views, models, etc.), but rather focus on the areas unique to RapidSMS itself, specifically messaging and the router.

15.1.1 What To Test

Let's start with an example. Say you've written a quiz application, `QuizMe`, that will send a question if you text the letter `q` to RapidSMS:

```
You: q
RapidSMS: What color is the ocean? Answer with 'q ocean <answer>'
You: q ocean red
RapidSMS: Please try again!
You: q ocean blue
RapidSMS: Correct!
```

Additionally, if no questions exist, the application will inform you:

```
You: q
RapidSMS: No questions exist.
```

While the application is conceptually simple, determining what and how to test can be a daunting task. To start, let's look a few areas that we could test:

- **Message parsing.** How does the application know the difference between `q` and `q ocean blue`? Will it be confused by other input, like `q ocean blue` or `quality`?
- **Workflow.** What happens when there aren't any questions in the database?
- **Logic testing.** Is the answer correct?

How to test these aspects is another question. Generally speaking, it's best practice, and conceptually the easiest, to test the smallest units of your code. For example, say you have a function to test if an answer is correct:

```
class QuizMeApp(AppBase):

    def check_answer(self, question, answer_text):
        """Return if guess is correct or not"""

        guess = answer_text.lower()
        answer = question.correct_answer.lower()
        return guess == answer
```

Writing a test that uses `check_answer` directly will verify the correctness of that function alone. With that test written, you can write higher level tests knowing that `check_answer` is covered and will only fail if the logic changes inside of it.

The following sections describe the various methods and tools to use for testing your RapidSMS applications.

15.2 Testing Methods

15.2.1 General Testing

RapidSMS provides a suite of test harness tools. Below you'll find a collection of `django.test.TestCase` extensions to make testing your RapidSMS applications easier.

RapidTest

The *RapidTest* class provides a simple test environment to analyze sent and received messages. You can inspect messages processed by the router and, if needed, see if messages were delivered to a special backend, `mockbackend`. Let's take a look at a simple example:

```
from rapidsms.tests.harness import RapidTest

class QuizMeStackTest(RapidTest):

    def test_no_questions(self):
        """Outbox should contain message explaining no questions exist"""
        self.receive('q', self.lookup_connections('1112223333')[0])
        self.assertEqual(self.outbound[0].text, 'No questions exist.')
```

In this example, we want to make sure that texting `q` into our application will return the proper message if no questions exist in our database. We use `receive` to communicate to the router and `lookup_connections` to create a `connection` object to bundle with the message. Our app will respond with a special message, `No questions exist`, if the database isn't populated, so we inspect the `outbound` property to see if it contains the proper message text. That's it! With just a few lines we were able to send a message through the entire routing stack and verify the functionality of our application.

class rapidsms.tests.harness.**RapidTest**(*methodName='runTest'*)
Inherits from *TestRouterMixin*, *LoginMixin*, TestCase.

Database Interaction

RapidTest provides flexible means to check application state, including the database. Here's an example of a test that examines the database after receiving a message:

```
from rapidsms.tests.harness import RapidTest
from quizme.models import Question, Answer

class QuizMeGeneralTest(RapidTest):

    def test_question_answer(self):
        """Outbox should contain question promt and answer should be recorded in database"""

        Question.objects.create(short_name='ocean',
                                text="What color is the ocean?",
                                correct_answer='Blue')
        msg = self.receive('q ocean blue', self.lookup_connections('1112223333')[0])
        # user should receive "correct" response
        self.assertEqual(self.outbound[0].text, 'Correct!')
        # answer from this interaction should be stored in database
        answer = Answer.objects.all()[0]
        self.assertTrue(answer.correct)
        self.assertEqual(msg.connection, answer.connection)
```

15.2.2 Application Logic

If you have application logic that doesn't depend on message processing directly, you can always test it independently of the router API. RapidSMS applications are just Python classes, so you can construct your app inside of your test suite. For example:

```
from django.test import TestCase
from rapidsms.router.test import TestRouter
from quizme.app import QuizMeApp

class QuizMeLogicTest(TestCase):

    def setUp(self):
        # construct the app we want to test with the TestRouter
        self.app = QuizMeApp(TestRouter())

    def test_inquiry(self):
        """Messages with only the letter "q" are quiz messages"""

        self.assertTrue(self.app.is_quiz("q"))

    def test_inquiry_whitespace(self):
        """Message inquiry whitespace shouldn't matter"""

        self.assertTrue(self.app.is_quiz(" q "))

    def test_inquiry_skip(self):
        """Only messages starting with the letter q should be considered"""
```

```
        self.assertFalse(self.app.is_quiz("quantity"))
        self.assertFalse(self.app.is_quiz("quality 50"))
```

This example tests the logic of `QuizMeApp.is_quiz`, which is used to determine whether or not the text message is related to the quiz application. The app is constructed with *`TestRouter`* and tests `is_quiz` with various types of input.

This method is useful for testing specific, low-level components of your application. Since the routing architecture isn't loaded, these tests will also execute very quickly.

15.2.3 Scripted Tests

You can write high-level integration tests for your applications by using the *`TestScript`* framework. *`TestScript`* allows you to write message *scripts* (akin to a movie script), similar to our example in the *What To Test* section above:

```
You: q
RapidSMS: What color is the ocean? Answer with 'q ocean <answer>'
You: q ocean blue
RapidSMS: Correct!
```

The main difference is the syntax:

```
1112223333 > q
1112223333 < What color is the ocean? Answer with 'q ocean <answer>'
1112223333 > q ocean blue
1112223333 < Correct!
```

The script is interpreted like so:

- **number > message-text**
  - Represents an incoming message from **number** whose contents is **message-text**
- **number < message-text**
  - Represents an outoing message sent to **number** whose contents is **message-text**

The entire script is parsed and executed against the RapidSMS router.

Example

To use this functionality in your test suite, you simply need to extend from *`TestScript`* or *`TestScriptMixin`* to get access to *`runScript()`*:

```
from rapidsms.tests.harness import TestScript
from quizme.app import QuizMeApp
from quizme.models import Question

class QuizMeScriptTest(TestScript):
    apps = (QuizMeApp,)

    def test_correct_script(self):
        """Test full script with correct answer"""

        Question.objects.create(short_name='ocean',
                                text="What color is the ocean?",
                                correct_answer='Blue')
        self.runScript("""
            1112223333 > q
```

```
            1112223333 < What color is the ocean? Answer with 'q ocean <answer>'
            1112223333 > q ocean blue
            1112223333 < Correct!
        """)
```

This example uses `runScript` to execute the interaction against the RapidSMS router. `apps` must be defined at the class level to tell the test suite which apps the router should load. In this case, only one app was required, `QuizMeApp`.

This test method is particularly useful when executing high-level integration tests across multiple RapidSMS applications. However, you're limited to the test script. If you need more fined grained access, like checking the state of the database in the middle of a script, you should use *General Testing*.

class `rapidsms.tests.harness.`**`TestScript`**(*methodName='runTest'*)
Inherits from *`TestScriptMixin`*, `TransactionTestCase`.

class `rapidsms.tests.harness.`**`TestScriptMixin`**
The scripted.TestScript class subclasses unittest.TestCase and allows you to define unit tests for your RapidSMS apps in the form of a 'conversational' script:

```
    from myapp.app import App as MyApp
    from rapidsms.tests.scripted import TestScript

    class TestMyApp (TestScript):
        apps = (MyApp,)
        testRegister = """
            8005551212 > register as someuser
            8005551212 < Registered new user 'someuser' for 8005551212!
        """

        testDirectMessage = """
            8005551212 > tell anotheruser what's up??
            8005550000 < someuser said "what's up??"
        """
```

This TestMyApp class would then work exactly as any other unittest.TestCase subclass (so you could, for example, call unittest.main()).

Inherits from *`TestRouterMixin`*.

`runScript`(*script*)
Run a test script.

Parameters **`script`** (*string*) – A multi-line test script. See *`TestScriptMixin`*.

class `rapidsms.tests.harness.scripted.`**`TestScriptMixin`**
Full name of *`rapidsms.tests.harness.TestScriptMixin`*.

15.2.4 Test Helpers

Below you'll find a list of mixin classes to help ease unit testing. Most of these mixin classes are used by the RapidSMS test classes for convenience, but you can also use these test helpers independently if needed.

CreateDataMixin

The `CreateDataMixin` class can be used with standard `TestCase` classes to make it easier to create common RapidSMS models and objects. For example:

```
from django.test import TestCase
from rapidsms.tests.harness import CreateDataMixin

class ExampleTest(CreateDataMixin, TestCase):

    def test_my_app_function(self):
        contact1 = self.create_contact()
        contact2 = self.create_contact({'name': 'John Doe'})
        connection = self.create_connection({'contact': contact1})
        text = self.random_string()
        # ...
```

class rapidsms.tests.harness.**CreateDataMixin**

Base test mixin class that provides helper functions to create data.

No superclasses.

create_backend(*data={}*)

Create and return RapidSMS backend object. A random `name` will be created if not specified in `data` attribute.

Parameters **data** – Optional dictionary of field name/value pairs to pass to the object's `create` method.

create_connection(*data={}*)

Create and return RapidSMS connection object. A random `identity` and `backend` will be created if not specified in `data` attribute.

Parameters **data** – Optional dictionary of field name/value pairs to pass to the object's `create` method.

create_contact(*data={}*)

Create and return RapidSMS contact object. A random `name` will be created if not specified in `data` attribute.

Parameters **data** – Optional dictionary of field name/value pairs to pass to the object's `create` method.

create_incoming_message(*data={}*)

Create and return RapidSMS IncomingMessage object.

create_outgoing_message(*data={}*, *backend=None*)

Create and return RapidSMS OutgoingMessage object. A random `template` will be created if not specified in `data` attribute.

Parameters **data** – Optional dictionary of field name/value pairs to pass to `OutgoingMessage.__init__`.

random_string(*length=255*, *extra_chars=u''*)

Generate a random string of characters.

Parameters

- **length** – Length of generated string.
- **extra_chars** – Additional characters to include in generated string.

random_unicode_string(*max_length=255*)

Generate a random string of unicode characters.

Parameters **length** – Length of generated string.

class rapidsms.tests.harness.base.**CreateDataMixin**
Full name for *rapidsms.tests.harness.CreateDataMixin*.

CustomRouterMixin

The CustomRouterMixin class allows you to override the *RAPIDSMS_ROUTER* and *INSTALLED_BACKENDS* settings. For example:

```
from django.test import TestCase
from rapidsms.tests.harness import CustomRouterMixin

class ExampleTest(CustomRouterMixin, TestCase)):

    router_class = 'path.to.router'
    backends = {'my-backend': {'ENGINE': 'path.to.backend'}}

    def test_sample(self):
        # this test will use specified router and backends
        pass
```

class rapidsms.tests.harness.**CustomRouterMixin**
Inheritable TestCase-like object that allows Router customization.

Inherits from *CreateDataMixin*.

backends = {}
Dictionary to override *INSTALLED_BACKENDS* during testing. Defaults to {}.

get_router()
get_router() API wrapper.

handlers = None
List to override RAPIDSMS_HANDLERS with, or if None, leave RAPIDSMS_HANDLERS alone

lookup_connections(*backend*, *identities*)
lookup_connections() API wrapper.

receive(*text*, *connection*, ***kwargs*)
A wrapper around the receive API. See *Receiving Messages*.

router_class = 'rapidsms.router.blocking.BlockingRouter'
String to override *RAPIDSMS_ROUTER* during testing. Defaults to 'rapidsms.router.blocking.BlockingRouter'.

send(*text*, *connections*, ***kwargs*)
A wrapper around the send API. See *Sending Messages*.

class rapidsms.tests.harness.router.**CustomRouterMixin**
Full name for *rapidsms.tests.harness.CustomRouterMixin*.

TestRouterMixin

TestRouterMixin extends CustomRouterMixin and arranges for tests to use the *rapidsms.router.test.TestRouter*.

class rapidsms.tests.harness.**TestRouterMixin**
Test extension that uses TestRouter

Inherits from *CustomRouterMixin*.

apps
A list of app classes to load, rather than `INSTALLED_APPS`, when the router is initialized.

clear_sent_messages()
Manually empty the outbox of mockbackend.

disable_phases = False
If *disable_phases* is True, messages will not be processed through the router phases. This is useful if you're not interested in testing application logic. For example, backends may use this flag to ensure messages are sent to the router, but don't want the message to be processed.

inbound
The list of message objects received by the router.

lookup_connections(*identities*, *backend='mockbackend'*)
A wrapper around the `lookup_connections` API. See *Connection Lookup*.

outbound
The list of message objects sent by the router.

sent_messages
The list of message objects sent to mockbackend.

class rapidsms.tests.harness.router.**TestRouterMixin**
Full name for *rapidsms.tests.harness.TestRouterMixin*.

TestRouter

The `TestRouter` can be used in tests. It saves all messages for later inspection by the test.

class rapidsms.router.test.**TestRouter**(**args*, ***kwargs*)
Router that saves inbound/outbound messages for future inspection.

Inherits from *BlockingRouter*.

inbound = None
List of all the inbound messages

outbound = None
List of all the outbound messages

receive_incoming(*msg*)
Save all inbound messages locally for test inspection

send_outgoing(*msg*)
Save all outbound messages locally for test inspection

DatabaseBackendMixin

The `DatabaseBackendMixin` helps tests to use the DatabaseBackend.

class rapidsms.tests.harness.**DatabaseBackendMixin**
Arrange for test to use the DatabaseBackend, and add a `.sent_messages` attribute that will have the list of all messages sent.

Inherits from *CustomRouterMixin*.

lookup_connections(*identities*, *backend='mockbackend'*)
lookup_connections wrapper to use mockbackend by default

sent_messages
Messages passed to backend.

LoginMixin

class rapidsms.tests.harness.**LoginMixin**
Helpers for creating users and logging in

login()
If not already set, creates self.username and self.password, otherwise uses the existing values. If there's not already a user with that username, creates one. Sets self.user to that user. Logs the user in.

class rapidsms.tests.harness.base.**LoginMixin**
Full name for *rapidsms.tests.harness.LoginMixin*.

Django TestCase

Some of these classes inherit from:

class django.test.testcases.**TestCase**

which is the full name for django.test.TestCase.

CHAPTER 16

Using Celery for Scheduling Tasks

You can use any scheduling mechanism supported by Django, but there are some advantages to using Celery. It's supported, scales well, and works well with Django. Given its wide use, there are lots of resources to help learn and use it. And once learned, that knowledge is likely to be useful on other projects.

You will also need to follow the setup instructions here if you are using RapidSMS's `CeleryRouter`.

16.1 History

After discussion, `rapidsms.contrib.scheduler` was removed in RapidSMS 0.12.0 in favor of adopting a wider Django community standard, Celery. Celery provides all the functionality of the previous built-in scheduler, but has the advantage of providing a more general solution for asynchronous task execution. Please see the 0.12.0 Release Notes for more details.

16.2 Celery versions

This documentation applies to Celery 3.0.x. Earlier or later versions of Celery might behave differently.

16.3 Introduction to Celery

The purpose of Celery is to allow you to run some code later, or regularly according to a schedule.

Why might this be useful? Here are a couple of common cases.

First, suppose a web request has come in from a user, who is waiting for the request to complete so a new page can load in their browser. Based on their request, you have some code to run that's going to take a while (longer than the person might want to wait for a web page), but you don't really need to run that code before responding to the web request. You can use Celery to have your long-running code called later, and go ahead and respond immediately to the web request.

This is common if you need to access a remote server to handle the request. Your app has no control over how long the remote server will take to respond, or the remote server might be down.

Another common situation is wanting to run some code regularly. For example, maybe every hour you want to look up the latest weather report and store the data. You can write a task to do that work, then ask Celery to run it every hour. The task runs and puts the data in the database, and then your Web application has access to the latest weather report.

A task is just a Python function. You can think of scheduling a task as a time-delayed call to the function. For example, you might ask Celery to call your function `task1` with arguments `(1, 3, 3)` after five minutes. Or you could have your function `batchjob` called every night at midnight.

We'll set up Celery so that your tasks run in pretty much the same environment as the rest of your application's code, so they can access the same database and Django settings. There are a few differences to keep in mind, but we'll cover those later.

When a task is ready to be run, Celery puts it on a queue, a list of tasks that are ready to be run. You can have many queues, but we'll assume a single queue here for simplicity.

Putting a task on a queue just adds it to a to-do list, so to speak. In order for the task to be executed, some other process, called a *worker*, has to be watching that queue for tasks. When it sees tasks on the queue, it'll pull off the first and execute it, then go back to wait for more. You can have many workers, possibly on many different servers, but we'll assume a single worker for now.

We'll talk more later about the queue, the workers, and another important process that we haven't mentioned yet, but that's enough for now, let's do some work.

16.4 Installing celery locally

Installing celery for local use with Django is trivial - just install django-celery:

```
$ pip install django-celery
```

16.5 Configuring Django for Celery

To get started, we'll just get Celery configured to use with `runserver`. For the Celery broker, which we will explain more about later, we'll use a Django database broker implementation. For now, you just need to know that Celery needs a broker and we can get by using Django itself during development.

In your Django `settings.py` file:

1. Add these lines:

```
1 import djcelery
2 djcelery.setup_loader()
3 BROKER_URL = 'django://'
```

The first two lines are always needed. Line 3 configures Celery to use its Django broker.

2. Add `djcelery` and `kombu.transport.django` to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    'djcelery',
    'kombu.transport.django',
    ...
)
```

djcelery is always needed. *kombu.transport.django* is the Django-based broker, for use mainly during development.

3. Create celery's database tables. If using South for schema migrations:

```
$ python manage.py migrate
```

Otherwise:

```
$ python manage.py syncdb
```

16.6 Writing a task

As mentioned before, a task can just be a Python function. However, Celery does need to know about it. That's pretty easy when using Celery with Django. Just add a `tasks.py` file to your application, put your tasks in that file, and decorate them. Here's a trivial `tasks.py`:

```
from celery import task

@task()
def add(x, y):
    return x + y
```

When `djcelery.setup_loader()` runs from your settings file, Celery will look through your `INSTALLED_APPS` for `tasks.py` modules, find the functions marked as tasks, and register them for use as tasks.

Marking a function as a task doesn't prevent calling it normally. You can still call it: `z = add(1, 2)` and it will work exactly as before. Marking it as a task just gives you additional ways to call it.

16.7 Scheduling it

Let's start with the simple case we mentioned above. We want to run our task soon, we just don't want it to hold up our current thread. We can do that by just adding `.delay` to the name of our task:

```
from myapp.tasks import add

add.delay(2, 2)
```

Celery will add the task to its queue (*"call myapp.tasks.add(2, 2)"*) and return immediately. As soon as an idle worker sees it at the head of the queue, the worker will remove it from the queue, then execute it:

```
import myapp.tasks.add

myapp.tasks.add(2, 2)
```

Import names

It's important that your task is always imported and refered to using the same package name. For example, depending on how your Python path is set up, it might be possible to refer to it as either `myproject.myapp.tasks.add` or `myapp.tasks.add`. Or from `myapp.views`, you might import it as `.tasks.add`. But Celery has no way of knowing those are all the same task.

`djcelery.setup_loader()` will register your task using the package name of your app in `INSTALLED_APPS`, plus `.tasks.functionname`. Be sure when you schedule your task, you also import it using that same name, or very confusing bugs can occur.

16.8 Testing it

16.8.1 Start a worker

As we've already mentioned, a separate process, the worker, has to be running to actually execute your Celery tasks. Here's how we can start a worker for our development needs.

First, open a new shell or window. In that shell, set up the same Django development environment - activate your virtual environment, or add things to your Python path, whatever you do so that you *could* use `runserver` to run your project.

Now you can start a worker in that shell:

```
$ python manage.py celery worker --loglevel=info
```

The worker will run in that window, and send output there.

16.8.2 Run your task

Back in your first window, start a Django shell and run your task:

```
$ python manage.py shell
>>> from myapp.tasks import add
>>> add.delay(2, 2)
```

You should see output in the worker window indicating that the worker has run the task:

```
[2013-01-21 08:47:08,076: INFO/MainProcess] Got task from broker: myapp.tasks.add[e080e047-b2a2-43a7-
[2013-01-21 08:47:08,299: INFO/MainProcess] Task myapp.tasks.add[e080e047-b2a2-43a7-af74-d7d9d98b02f
```

16.9 An Example

Earlier we mentioned using Celery to avoid delaying responding to a web request. Here's a simplified Django view that uses that technique:

```
# views.py

def view(request):
    form = SomeForm(request.POST)
    if form.is_valid():
        data = form.cleaned_data
        # Schedule a task to process the data later
        do_something_with_form_data.delay(data)
    return render_to_response(...)

# tasks.py

@task
def do_something_with_form_data(data):
    call_slow_web_service(data['user'], data['text'], ...)
```

16.10 Troubleshooting

It can be frustrating trying to get Celery tasks working, because multiple parts have to be present and communicating with each other. Many of the usual tips still apply:

- Get the simplest possible configuration working first.
- Use the python debugger and print statements to see what's going on.
- Turn up logging levels (e.g. `--loglevel debug` on the worker) to get more insight.

There are also some tools that are unique to Celery.

16.10.1 Eager scheduling

In your Django settings, you can add:

```
CELERY_ALWAYS_EAGER = True
```

and Celery will bypass the entire scheduling mechanism and call your code directly.

In other words, with `CELERY_ALWAYS_EAGER = True`, these two statements run just the same:

```
add.delay(2, 2)
add(2, 2)
```

You can use this to get your core logic working before introducing the complication of Celery scheduling.

16.10.2 Peek at the Queue

As long as you're using Django itself as your broker for development, your queue is stored in a Django database. That means you can look at it easily. Add a few lines to `admin.py` in your application:

```
from kombu.transport.django import models as kombu_models
site.register(kombu_models.Message)
```

Now you can go to `/admin/django/message/` to see if there are items on the queue. Each *message* is a request from Celery for a worker to run a task. The contents of the message are rather inscrutable, but just knowing if your task got queued can sometimes be useful. The messages tend to stay in the database, so seeing a lot of messages there doesn't mean your tasks aren't getting executed.

16.10.3 Check the results

Anytime you schedule a task, Celery returns an AsyncResult object. You can save that object, and then use it later to see if the task has been executed, whether it was successful, and what the result was.

```
result = add.delay(2, 2)
...
if result.ready():
    print "Task has run"
    if result.successful():
        print "Result was: %s" % result.result
    else:
        if isinstance(result.result, Exception):
            print "Task failed due to raising an exception"
            raise result.result
```

```
        else:
            print "Task failed without raising exception"
else:
    print "Task has not yet run"
```

16.11 Periodic Scheduling

Another common case is running a task on a regular schedule. Celery implements this using another process, celerybeat. Celerybeat runs continually, and whenever it's time for a scheduled task to run, celerybeat queues it for execution.

For obvious reasons, only one celerybeat process should be running (unlike workers, where you can run as many as you want and need).

Starting celerybeat is similar to starting a worker. Start another window, set up your Django environment, then:

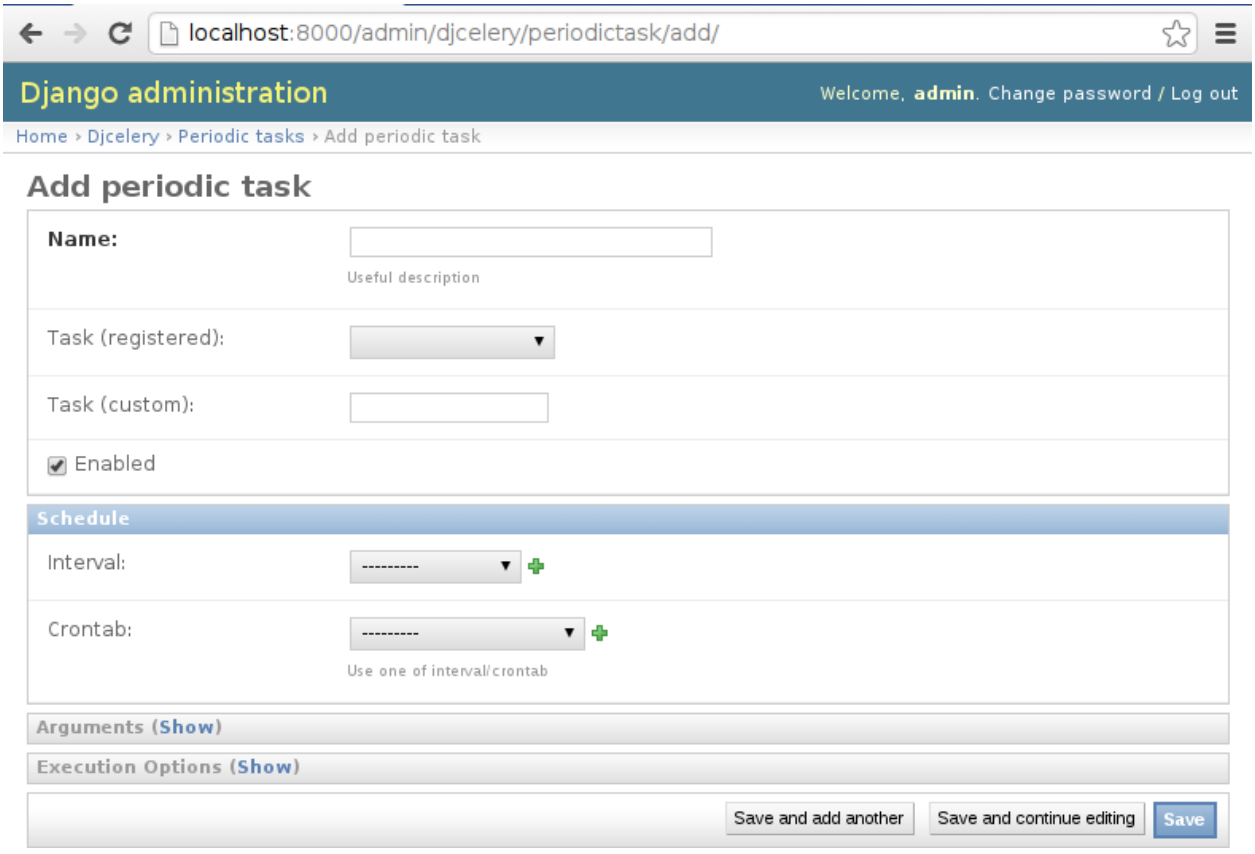
```
$ python manage.py celery beat
```

There are several ways to tell celery to run a task on a schedule. We're going to look at storing the schedules in a Django database table. This allows you to easily change the schedules, even while Django and Celery are running.

Add this setting:

```
CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler'
```

You can now add schedules by opening the Django admin and going to /admin/djcelery/periodictask/. Here's what adding a new periodic task looks like:



Name Any name that will help you identify this scheduled task later.

Task (registered) This should give a choice of any of your defined tasks, as long as you've started Django at least once after adding them to your code. If you don't see the task you want here, it's better to figure out why and fix it than use the next field.

Task (custom) You can enter the full name of a task here (e.g. `myapp.tasks.add`), but it's better to use the registered tasks field just above this.

Enabled You can uncheck this if you don't want your task to actually run for some reason, for example to disable it temporarily.

Interval Use this if you want your task to run repeatedly with a certain delay in between. You'll probably need to use the green "+" to define a new schedule. This is pretty simple, e.g. to run every 5 minutes, set "Every" to 5 and "Period" to minutes.

Crontab Use crontab, instead of *Interval*, if you want your task to run at specific times. Use the green "+" and fill in the minute, hour, day of week, day of month, and day of year. You can use "*" in any field in place of a specific value, but be careful - if you use "*" in the Minute field, your task will run every minute of the hour(s) selected by the other fields. Examples: to run every morning at 7:30 am, set Minute to "30", Hour to "7", and the remaining fields to "*".

Arguments If you need to pass arguments to your task, you can open this section and set *args and **kwargs.

Execution Options Advanced settings that we won't go into here.

16.11.1 Default schedules

If you want some of your tasks to have default schedules, and not have to rely on someone setting them up in the database after installing your app, you can use Django fixtures to provide your schedules as initial data for your app.

- Set up the schedules you want in your database.
- Dump the schedules in json format:

```
$ python manage.py dumpdata djcelery --indent=2 --exclude=djcelery.taskmeta >filename.json
```

- Create a `fixtures` directory inside your app
- If you never want to edit the schedules again, you can copy your json file to `initial_data.json` in your fixtures directory. Django will load it every time `syncdb` is run, and you'll either get errors or lose your changes if you've edited the schedules in your database. (You can still add new schedules, you just don't want to change the ones that came from your initial data fixture.)
- If you just want to use these as the initial schedules, name your file something else, and load it when setting up a site to use your app:

```
$ python manage.py loaddata <your-app-label/fixtures/your-filename.json
```

16.12 Hints and Tips

16.12.1 Don't pass model objects to tasks

Since tasks don't run immediately, by the time a task runs and looks at a model object that was passed to it, the corresponding record in the database might have changed. If the task then does something to the model object and saves it, those changes in the database are overwritten by older data.

It's almost always safer to save the object, pass the record's key, and look up the object again in the task:

```
myobject.save()
mytask.delay(myobject.pk)

...


@task
def mytask(pk):
    myobject = MyModel.objects.get(pk=pk)
    ...
```

16.12.2 Schedule tasks in other tasks

It's perfectly all right to schedule one task while executing another. This is a good way to make sure the second task doesn't run until the first task has done some necessary work first.

16.12.3 Don't wait for one task in another

If a task waits for another task, the first task's worker is blocked and cannot do any more work until the wait finishes. This is likely to lead to a deadlock, sooner or later.

If you're in Task A and want to schedule Task B, and after Task B completes, do some more work, it's better to create a Task C to do that work, and have Task B schedule Task C when it's done.

16.13 Next Steps

Once you understand the basics, parts of the Celery User's Guide are good reading. I recommend these chapters to start with; the others are either not relevant to Django users or more advanced:

- Tasks
- Periodic Tasks

16.14 Using Celery in production

The Celery configuration described here is for convenience in development, and should never be used in production.

The most important change to make in production is to stop using *kombu.transport.django* as the broker, and switch to RabbitMQ or something equivalent that is robust and scalable.

CHAPTER 17

Packaging your RapidSMS application for re-use

If you'd like others to be able to use your application, you'll want to package it and publish it on PyPI.

You will package and publish your RapidSMS application in the same way you would any other Django application. Django provides excellent documentation on packaging your Django app, so we won't try to write the same thing here.

We recommend using at least the following classifiers on your package:

```
Framework :: Django
Intended Audience :: Developers
Programming Language :: Python
Topic :: Communications
Topic :: Software Development :: Libraries :: Python Modules
```

Depending on your project, also consider:

```
Operating System :: OS Independent
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Environment :: Web Environment
```

You'll also need to give your package a license that allows others to use it. RapidSMS uses the BSD license and we recommend it if you don't have a strong preference for another license.

CHAPTER 18

Provisioning Servers & Deploying Your Project

Provisioning are the steps required to prepare a server for a RapidSMS project installation. *Deployment* is the process of continually syncing code to a production environment as changes are made and tested in a development environment.

RapidSMS projects can easily be installed in many ways. Provisioning & Deployment are a large topics, contain many right answers and are largely dependent on your application requirements. Our goal is not to provide the best solution or a configuration that will work on any project. We only want to provide you with the proper resources to make the best decisions.

Document sane defaults. While there are many installation methods, there's a common denominator of best practices that all production RapidSMS sites should follow (don't use DEBUG = True). We will document a concise list of best practices.

Example templates. There are many options to consider: a bare metal server, cloud VM, platform as a service (PaaS). We don't want to bless any single particular method, but we believe that linking to sample configurations for a small subset of these will provide a solid foundation and starting point for deploying your own application.

We can look at the overall production installation process in four parts:

- Deciding where to run your application - *Planning for a Provisioning & Deployment*.
- Preparing the server(s) to run your application - *Provisioning*.
- Deployiong and updating your application to the server(s) - *Deploying your Application*.
- Scaling up when your server can't handle the traffic anymore - *Scaling*.

When deploying RapidSMS, you might also need to consider *Gateways and Telecom Operators*.

Note: Even if you don't read anything else, the main things are:

- Stop using runserver and switch over to Apache or your real server of choice.
- Use a real database (not SQLite)
- Turn off `DEBUG`!
- Follow the guidelines in *What to Provision*.

You can find community-contributed examples on the GitHub wiki.

18.1 Outline

18.1.1 Planning for a Provisioning & Deployment

There are three major types of hosting: application physical servers, hosted virtual machines, and platform as a service providers. Though running and maintaining a physical server has become less common recently, there are still RapidSMS installation situations where this is the best options. For example, there are times when a physical cellular modem is the best solution for sending and receiving SMS messages in a region. In this case, it can make sense to have a physical server plugged into a modem where the modem has service.

	Low Cost	Simplicity	Physical Modem	Customizable	Portability
Physical Server					
Hosted VM					
PaaS					

Shared and dedicated virtual machines (VMs) are becoming more and more common. In this hosting environment, the server and network configuration and maintenance are the responsibility of the hosting company. Maintaining a physical computer means worrying about RAM and hard drive failures, but this is not the case with VMs. Beyond this, many VM providers are providing APIs which make it possible to programmatically create a new server with an API request to the VM hosting provider. Once the VM has started, you are still responsible for installing an operating system and configuring all of the various software packages and services.

Further along on the spectrum of hosting providers are Platform as a Service (PaaS) providers. These providers not only provide an API for turning on a new server, but their servers come fully configured out of the box with a number of running services or access to shared services. For example, you will not need to install and configure your own web server, database server, or queue server. Each provider has its own simple web site and command line tool that help you create new services and deploy your application either from a software repository or your development system. Typically in this environment, you pay per service rather than per server.

As you travel up the spectrum from physical server to PaaS you typically get less maintenance at the cost of a higher price and reduced flexibility in your configuration. Furthermore, when you start using a PaaS, your deployment workflows will become specific to that PaaS. That is, the command line tools that you use to deploy your application will only work with that single provider. This reduces the portability of your configuration and locks you into one provider. Also, the reduced flexibility means that you won't have total control over the versions of software installed on your server and won't be able to install custom services. PaaS providers are typically more expensive than VM providers since they provide an extra layer of abstractions above the bare virtual hardware. But, if your application can be supported by a PaaS provider, it means not having to worry about security upgrades and initial installation for services.

18.1.2 Running on a Virtual Machine

Running RapidSMS projects on a virtual machine from a provider on the Internet is a popular choice. You get complete control over the system as if you were using your own hardware, but don't have to worry about any hardware-related problems like disk failures, providing reliable power, getting a good network connection, etc.

Using a VM over the Internet

The experience is similar with most VM providers. After setting up an account, you can create a new VM and select which of a number of installed operating system images it is created with. The provider will (in essence) install that operating system, configure the system to communicate over their network, and set up a way for you to login to the system.

Typically for our purposes, you'd install a server Linux image, with command line access only. Once you're logged in, you can do anything on the system that you could do while logged into your own hardware with a command line interface.

It's always possible to make a mistake when changing the system configuration and end up not able to login over the network anymore. The better providers give you an alternate way to get into your system so you can correct that yourself. With other providers, you might have to make a support request and wait for someone to fix it manually, perhaps with an additional charge.

Choosing a provider

There are many providers of VMs on the Internet. Amazon is probably the biggest. Others like Linode and Rackspace are also popular.

There are two main considerations when choosing a VM provider, reliability and connectivity/location. Reliability is the most important - if your system isn't running, whether it is well-connected to the Internet is moot. Some providers publish uptime statistics or offer service level guarantees, or you could just go with one of the biggest, best-known providers.

Location is another criterion. If your users are accessing your app over the web, they'll likely get better response times if the server is at least on the same continent as they are (though not always).

Cost is always a factor, but it should not be the overriding criterion. There are some very low-priced VM providers, but typically they run many more VMs on the same hardware so that performance is poor, and they offer poor or no service.

Here are a few of the most popular VM providers:

- Amazon EC2 has nine regions: US East (Northern Virginia), US West (Oregon), US West (Northern California), EU (Ireland), Asia Pacific (Singapore), Asia Pacific (Tokyo), Asia Pacific (Sydney), South America (Sao Paulo), and AWS GovCloud.
- Linode has six data centers: Tokyo, London, Newark, Atlanta, Dallas, and Fremont.
- Rackspace doesn't appear to have information on their web site about their locations.

18.1.3 Running on a PaaS Provider

In choosing to run a project on a PaaS provider, you are choosing simplicitiy over ability to easily customize and transfer your installation. Each of these companies provide a custom command line tool for automatically configuring and deploying an application.

Choosing a provider

There are a number of PaaS providers who support Django web applications. Most of these will work fine for supporting RapidSMS. It is important to consider the availability and cost of extra services since you will be relying on the services provided out of the box and there is little room for customization.

It should also be noted that there have been a number of PaaS solutions that are no longer operating businesses. Since there is less portability, it is important to consider the longevity of the provider when finalizing a decision. Also, with respect to portability, since you are choosing to invest in a single company's tools, the documentation and support that are provided by the company must be vetted before deciding to go down a particular path.

Finally, another important consideration to take into account is the command line tool provided by the PaaS provider itself. With a virtual machine, for example, you can choose from a number of tools to put on top of the VM to help provision a new instance while with PaaS, you are locked in to the tools decided on by the provider.

Here's a short list of popular PaaS providers that support Django:

- Heroku
- gondor.io
- dotCloud

Examples

The RapidSMS wiki has a page with links to examples of how people provision and deploy RapidSMS applications.

18.1.4 Provisioning

Provisioning is making sure the systems where you're going to run your RapidSMS site are ready to deploy to. After a system is installed, typically you need to create users, grant permissions, and install and configure tools such as a database, a web server, a cache, a message broker, etc.

Another way to look at it is that provisioning is things you only need to do once per system, like installing a database server, while deploying includes the things you have to do separately for each app or site you deploy, like creating a virtual environment for it.

You might also look at provisioning as all the things you have to do for yourself if you're not using a PaaS to host your site.

Above all else, you want your provisioning to be repeatable. Use tools to automate it so that you don't waste a lot of time tracking down problems that turn out to be the result of a missing package on one server, or an incorrect configuration, or incorrect file permission, and so forth.

Having automated provisioning can also be very useful to be able to easily provision new systems. For example, you can set up a new test system, or add another staging server, or another production server.

We're going to look at two aspects of provisioning:

- *Ways to Provision*
- *What to Provision*

18.1.5 Ways to Provision

Manual provisioning

You can provision manually, installing things as you discover you need them, and tweaking the configuration until things work. This is often how people's development systems end up getting provisioned.

If you take careful notes as you provision manually, then the next time you have to do it, you can follow the notes and do it much more quickly and with fewer errors. You can also share the notes so your co-workers can benefit.

Scripting

The next step is putting the provisioning commands into a script of some kind. Typically this is when you have to start worrying more about what platform you're working on. For example, the commands to install packages on Debian-based Linuxes are different from those on Redhat or Gentoo-based Linuxes, so your script either needs to assume some base platform, or it suddenly gets a lot more complicated.

At this level, you can use tools like Fabric to help organize your provisioning commands into tasks and execute them remotely. However, Fabric does not help with issues of provisioning like how to install a package or create a user.

For an example of this, you might look at the Django project template offered by Caktus. Its fabfile, with the help of Argyle, does provisioning on Ubuntu systems. You can see explicit commands to do things like create symbolic links and change ownership of files.

Provisioning tools

After doing that for a while, you'll probably wish there were some tools you could use to save you from having to understand every detail of setting up your system, and how administering your Ubuntu system is different from a Redhat system.

You might start with something like Blueprint. It takes a different approach than most of the other tools we'll mention. With Blueprint, instead of writing a specification for what provisioning needs to be done, you run a Blueprint tool on a system that you've provisioned already and it records the state of the system: which packages have been added to the defaults, which configuration files have been changed or added and what their contents are, etc.

Then Blueprint provides multiple ways to recreate that provisioned system, including generating a shell script that will do it, or exporting a set of configuration files for several other provisioning tools such as Puppet and Chef. There's a lot of flexibility and room to finetune how Blueprint works; this is just the surface.

Beyond Blueprint there are tools like Chef, Puppet, or SaltStack. These provide full languages in which to specify how many systems should be configured, and tools to apply and maintain the configurations. The cost of this power, of course, is complexity. Getting one of these provisioning tools installed and learning to use it will require a large investment of time.

Examples

The RapidSMS wiki has a page with links to examples of how people provision and deploy RapidSMS applications.

Recommendations on Implementing Provisioning

Unfortunately, there's no simple solution to implementing repeatable provisioning. Provisioning is a very complicated task.

Tools can help, but you still need a pretty detailed knowledge of system administration to make it work right.

One consideration is how large your problem is. For people who have to manage more than a dozen or so systems, a full-featured tool like Chef or Puppet is essential, and the time spent learning it thoroughly will be repaid many times over.

If you only have a few Django apps, and they're fairly conventional in terms of provisioning requirements, you should consider a *PaaS*. They handle the provisioning for you. The monthly bills will be higher, but your costs in time spent getting provisioning working and keeping it working will be much lower.

If your needs fall somewhere in the middle, or funding isn't available for a PaaS, then you'll need to consider your needs and your level of expertise, try out some of the available approaches, and make the best decision you can. The RapidSMS mailing list is a good resource; you can present your situation there and ask for advice.

18.1.6 What to Provision

A Good Basic Setup

One of the intimidating things about deploying a Django application is the number of decisions that need to be made.

Here are a recommended set of choices for deploying Django apps. Most of these choices have alternatives, and some are decided for you if you're using a PaaS, but in the absence of a reason to use an alternative, these should work reasonably well in most cases.

Operating System Ubuntu LTS server: Ubuntu because it's stable, it's free, and it's popular among developers, so any Django or Python software is likely to work well on it. LTS because of the long support guarantee, and server since we don't need costly graphical desktop environments on our servers.

Django version The latest released version: because it will be supported by security fixes for the longest time, compared to any older version, and Django has a history of putting out pretty stable releases. (Maybe wait a couple of weeks after a new release to see if there's a .1 release.)

Database PostgreSQL: because it and MySQL are the most popular free databases, so it's well supported, and MySQL is lacking some features. For example, MySQL cannot perform schema changes in transactions, so if a schema migration fails in the middle, your data could be left in an indeterminate state.

Sometimes there are reasons to use another database than PostgreSQL. Just be sure not to use SQLite, even though it's very easy to set up. It's not suited for production use.

Web server Apache or nginx: both work well as front ends for Django applications.

It is important to have a web server handle incoming requests, rather than having them go directly to the Django application, for a couple of reasons:

- Web servers are designed to efficiently process the load of incoming requests, and deal with the wide variety of web clients. That lets the WSGI server focus on hosting the Django application.
- The Django application should never be used to serve static files in production. If the static files are not being served by another system, the web server is used to either serve the static files, or proxy the requests to something other than the Django application.

WSGI server mod_wsgi with Apache, or gunicorn: Either of these work well.

The WSGI server provides a Python process to run your Django application.

mod_wsgi can be convenient if you are already be using Apache on the server for static files. Gunicorn is easier to configure and run.

In any case, never use *runserver* in production. It is not secure, and performs poorly.

Message queue server RabbitMQ: because it's stable and popular. Redis is also commonly used as a message queue server.

Schema migration South: because it's really the only choice available for schema migrations in Django.

RapidSMS Router `DatabaseRouter`: because it has the most features.

Asynchronous task scheduler Celery: see *Using Celery for Scheduling Tasks*.

Application settings Applications settings should be shared between different environments as much as possible. Keeping settings files available to all developers through your version control system keeps all of the developers on the same page and allows new developers to get started on your project quickly. The Django Book covers managing settings.py files between environments within projects.

Server Setup Recommendations

These are some recommendations for setting up your server.

Fail2ban install to detect and block some intrusion attempts

Firewall Block any incoming traffic that isn't needed by your application or server. Ubuntu provides the ufw tool which makes this easy.

Logwatch Logwatch will check your system logs daily and email you a daily report. This is helpful for spotting unusual activity.

Automatic security updates You can set up Ubuntu to automatically install security-related updates.

ntp use ntp to keep system clock up to date

18.1.7 Deploying your Application

RapidSMS can easily be deployed in many ways. Deployment is a large topic, contains many right answers and is largely dependent on your application requirements. Our goal is not to provide the best solution or a configuration that will work on any application. We only want to provide you with the proper resources to make the best decisions.

Typical Deployment Steps

In order to deploy, you need to accomplish a number of things.

Initially:

- Establish a virtual environment
- Install Python dependencies
- Install project code
- Establish settings specific for that server (secret keys, passwords, location of database, etc.)
- Sync database
- Collect static files
- Start processes (web server, workers, etc)

On each deploy:

- Update Python dependencies
- Update project code
- Apply migrations to database
- Collect static files
- Restart processes

You might also want to:

- Backup or download the database
- Restore or upload the database
- Backup/restore user-uploaded files

Different environments

The options for accomplishing these things depend in part on the server environment.

Non-PaaS

The Django documentation provides some advice about how to run a Django application like RapidSMS in production.

One approach you'll often see when deploying to your own hardware or to a virtual machine is using Fabric to implement deploy commands that a developer can use. A developer might type `fab staging deploy` to update the code on the staging server, or `fab production setup_server` to provision the production server. To help you write these commands, Fabric provides a library of methods for running commands on your remote server, uploading files, etc.

PaaS

If you're using a PaaS, your provider takes care of setting up your application on their servers, and will provide tools and documentation about how to deploy your application.

Examples

The RapidSMS wiki has a page with links to examples of how people provision and deploy RapidSMS applications.

18.1.8 Scaling

If your application is successful, you might find that your single server cannot keep up anymore. Responses might become slow, or operations might just not complete.

There are many steps you can take to scale up your implementation so that it can handle more traffic.

Rather than going into any more detail here, we'd like to direct you to the Scaling section of the Deployment chapter of the Django Book, which you can read online. This section walks you step-by-step through the changes you can make as your traffic grows in order to keep up.

18.1.9 Gateways and Telecom Operators

Before setting up your SMS service, consider a few questions:

- Do you have technical staff and just want to try out an SMS service on a trial basis? Or do you want something robust from the start? This will determine whether you begin your exploration with a tethered phone/GSM modem or start looking immediately at SMS gateways.
- Do you need to offer your service via a) a shortcode, or b) is a regular phone number sufficient? If you want a shortcode, which mobile operators do you need to support? Do you need or expect your service to expand to multiple countries? Acquiring a shortcode is a time-consuming process, which needs to be persued individually with each operator in each country.
- If a longcode is sufficient, do you need to minimize cost to your end users? This determines whether you need to support a local number or whether you can work with international SMS gateways and international numbers.

To implement an SMS service, you can take the following approaches:

Set up a GSM Modem

This can be as simple as tethering a mobile phone to a laptop. More commonly, you'll purchase a dedicated GSM modem (which has better performance than a regular mobile phone) and attach it to a server somewhere. In the modem you'll place a local GSM-enabled SIM card, run the application on your server, and you're done.

Pros:

- Easy and fast to setup

Cons:

- Not a good long-term solution. Managing a GSM modem requires more knowledge than just managing traditional servers on a network, and most small or non-technical organizations will not have the capacity to ensure 99% uptime. This may be a viable solution for a larger organization that already hosts its own web servers and has dedicated technical staff.
- Shortcode not possible

Local SMS Gateway

If you can find a reliable SMS gateway locally, this may be your best option, as they can provide local numbers, facilitate the acquisition of local short codes, and ideally provide competitive prices. Whether this is easy varies from country to ountry. In many countries, it is difficult and time-consuming to find a local provider who is timely and reliable.

Pro:

- Typically local gateways are in the best position to facilitate acquisition of local shortcodes among multiple mobile operators

International SMS Gateway

There are many reputable international SMS gateways, although their service and availability within a particular country can vary greatly and should always be tested before launch.

Pros:

- Can provide strong technical infrastructre and volume discounts

Cons:

- Texting an international number is more expensive for end users
- Sending text messages internationally can be expensive
- Shortcodes not possible

Host a local SIM with an international SMS gateway

One common solution we use is to take a local SIM card from a given country, enable roaming, and then ship it to an international SMS gateway. This gives us many benefits of the above two solutions without the drawbacks.

Pros:

- Users can text a local number cheaply
- Can provide strong technical infrastructre and volume discounts

Cons:

- Depends on the roaming infrastructure for a given network
- Shortcode not possible

Partnering with a Telecom Operator

Most SMS services are offered through third-party SMS gateways, whose business involves making it easier for third parties to offer services via SMS. Although it is rare, from time to time an organization can be lucky enough to form a strong partnership directly with a telecom operator.

Pros:

- Can tap in directly to local telecom infrastructure. Less chance of messages being lost.
- Strong partnership with telco can result in other significant benefits, such as discounts, additional services, and better accounting/reporting.

Cons:

- Telcos tend to be larger, often international organizations, and like any such organization, can move very slowly.
- Most SMS services are not large enough to warrant the attention of large telcos.
- If a shortcode is needed, still need to negotiate these with other telcos, either directly or through a gateway service.

Terminology

Gateway: A gateway is a web service which provides access to particular services. A website can interface with an SMS gateway over the internet in order to send and recieve SMS.

CHAPTER 19

Developing RapidSMS

RapidSMS is an open source project. That means you can get the source code and change it to suit yourself. You can offer your changes back to the project. You can help fix bugs, write new features, and even help run the project.

This section of the documentation is about working on RapidSMS. There are technical details, and some of the standards we follow for this code.

See *The RapidSMS Community* for more about how our development community works.

- *Getting the code for development*
- *Submit a pull request*
- *Coding standards and best practices*
- *Writing documentation*
- *RapidSMS core test suite*
- *RapidSMS Release Checklist*

19.1 Getting the code for development

Here's how you can get the RapidSMS source code and make changes to it.

19.1.1 Git

The RapidSMS project uses Git to manage its source code. If you're not familiar with Git already, you'll find it useful knowledge for more than working on RapidSMS. Git is an incredibly popular source control tool, and there is a huge amount of documentation on the net ranging from introductory tutorials to in-depth reference material.

19.1.2 Clone the repository

Clone the RapidSMS source repository from Github:

```
$ git clone git://github.com/rapidsms/rapidsms.git
Cloning into 'rapidsms'...
remote: Counting objects: 25262, done.
remote: Compressing objects: 100% (8696/8696), done.
remote: Total 25262 (delta 15920), reused 24482 (delta 15498)
Receiving objects: 100% (25262/25262), 5.20 MiB | 1.09 MiB/s, done.
Resolving deltas: 100% (15920/15920), done.
```

You'll now have a new `rapidsms` subdirectory containing the code.

19.1.3 Environment

Install virtualenv and virtualenvwrapper (see *Using virtualenv*).

Change into the `rapidsms` directory:

```
$ cd rapidsms
$
```

Create a virtual environment to work in and activate it:

```
$ mkvirtualenv --no-site-packages rapidsms
...
$
```

19.1.4 Master branch

The default branch in the RapidSMS repository is `develop`, because that's the branch used when working on new features for an upcoming release, and so most developers use it a lot.

But let's switch to the stable branch, `master`, for now, so we can run the tests and verify that we have things set up right.

```
$ git checkout master
Branch master set up to track remote branch master from origin.
Switched to a new branch 'master'
$
```

19.1.5 Requirements

Install RapidSMS's requirements using distribute's develop command:

```
$ python setup.py develop
[lots of output omitted]
$
```

19.1.6 Tests

Verify that everything is okay by running RapidSMS's tests.

```
$ tox
[lots of output omitted]
_____________________________________ summary _____________________________________
  py26-1.4.X: commands succeeded
  py26-1.5.X: commands succeeded
  py26-trunk: commands succeeded
  py27-1.4.X: commands succeeded
  py27-1.5.X: commands succeeded
  py27-trunk: commands succeeded
  congratulations :)
$
```

This takes a while the first time – over 8 minutes on my computer. But after that, the environments are already set up and it'll run much faster. On my computer, subsequent tests take less than 30 seconds.

The code on the RapidSMS master branch should always pass the tests. If anything fails, review these instructions, and if they still fail, ask on IRC or the rapidsms-dev mailing list.

19.1.7 Work on a branch

When you're ready to start making changes, you'll want to create a new branch. You have a choice to base your branch on the `master` or `develop` branch. The tip of the `master` branch is always the latest released code. It's stable, but does not include any changes currently under development for the next release.

The `develop` branch contains changes that are ready for the next release. It should also be pretty stable, because all changes are developed on other branches and not merged into `develop` until they appear to be ready, but there's a bit more chance of there being something broken in `develop`.

It's probably a good idea to base your branch on `develop` if possible, because if you work from master, there might be changes already in develop that your work won't take into account.

19.2 Submit a pull request

To ask for a change you've made in your own RapidSMS repository to be merged into the official repository, you make a pull request.

If you're already familiar with Github and pull requests, here's all you need to know:

1. Sign a *Contributor License Agreement*.

2. Open a pull request against the `develop` branch of the RapidSMS repository (https://github.com/rapidsms/rapidsms).

Otherwise, here are the details.

19.2.1 Branch

Work on a branch from the *develop* branch. RapidSMS uses the Git Flow system to manage branches and releases. If you're using the gitflow tool, you can start your branch with:

```
$ git flow feature start NAME
Switched to a new branch 'feature/NAME'

Summary of actions:
- A new branch 'feature/NAME' was created, based on 'develop'
- You are now on branch 'feature/NAME'

Now, start committing on your feature. When done, use:

     git flow feature finish NAME

$
```

If you're not using the gitflow tool, create a new branch from develop and switch to it:

```
$ git branch feature/NAME develop
$ git checkout feature/NAME
```

```
Switched to branch 'feature/NAME'
$
```

19.2.2 Github

You'll need a Github account to proceed. They're free.

19.2.3 Fork the repository

Once you have a Github account, login, then go to the RapidSMS repository on Github and create a fork by clicking the `Fork` button near the top right of the page.

19.2.4 Add a remote

We'll assume you already have a local clone of the repository that you've been working in, so all you need to do is add a new remote definition pointing to your new fork. You might choose to name your new remote definition using your Github username, to keep it distinguished from the remote pointing at the official repository.

```
$ git remote add username git@github.com:username/rapidsms.git
$ git remote -v
username    git@github.com:username/rapidsms.git (fetch)
username    git@github.com:username/rapidsms.git (push)
origin      git://github.com/rapidsms/rapidsms.git (fetch)
origin      git://github.com/rapidsms/rapidsms.git (push)
$
```

19.2.5 Push your change to your own repository

Your change needs to be on Github before you can open a pull request against the RapidSMS code. Unless you have RapidSMS commit privileges, you'll need to upload your change to your own fork of the repository.

Assuming your branch name is feature/NAME:

```
$ git push -u username feature/NAME
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:username/rapidsms.git
 * [new branch]      feature/NAME -> feature/NAME
Branch feature/NAME set up to track remote branch feature/NAME from username.
$
```

19.2.6 Open a pull request

Go to your fork of the RapidSMS repository on the Github web site (https://github.com/username/rapidsms/).

Click the `Pull request` button near the top center of the page.

Check the settings:

base repo rapidsms/rapidsms

base branch develop

head repo username/rapidsms (your repo)

head branch feature/NAME (your branch)

Enter an informative name and description for your pull request. By default, Github will try to pull these from your commit messages, but often you can improve them.

Your pull request description is your chance to convince people that your change is worthwhile and will benefit RapidSMS. Here are some things to consider addressing:

- why is this change needed
- why will this be useful to general RapidSMS users (not just you)
- what change was made
- why that change was made, as opposed to some other way of achieving the same ends
- what testing has been done
- whether this change has already been used in real RapidSMS applications
- etc.

Switch to the `Files Changed` tab and check that the changes shown are what you expect. If not, go back and check that you've committed the changes you intended on your branch, that you pushed it to your repo, and that the pull request settings are correct for your repo and branch.

When everything looks right, switch back to the `New Pull Request` tab and click the green `Send pull request` button in the lower right, below the description box.

19.3 Coding standards and best practices

We follow these practices when developing RapidSMS code:

1. Work on a branch off the `develop` branch.
2. Follow PEP8 style conventions. Use 4 spaces instead of tabs.
    - To learn more about writing 'pythonic' code, check out The Hitchhiker's Guide To Python's Code Style guide
    - Tip: Configure your development environment for python to make your life a bit easier.
3. Run the *PEP 8 adherence tool*.
4. Use CapitalizedCase for class names, underscored_words for method names.
5. Code using `os.path` must be Windows and 'NIX friendly. For example, don't use backslashes (`\`) as path separators.
6. Be sure every class and method has docstrings.
7. Use *Python logging* whenever an error or exception occurs. Optionally include debug-level logging.
8. Write a test which shows that the bug was fixed or that the feature works as expected.
9. Run the *RapidSMS core test suite* to make sure nothing unexpected broke. We only accept pull requests with passing tests.
10. Write new or update existing *documentation* to describe the changes you made.
11. Add the change to the release notes document for the next release. The release notes should focus on the effects on existing users, particularly if it will require them to make changes.
12. Submit a pull request and get reviews before merging your changes, even if you have authority to merge the changes yourself.

13. Sign the *Contributor License Agreement*.

19.4 Writing documentation

We believe RapidSMS needs to treat our documentation like we treat our code. It's what you're reading now and is generally the first point of contact for new developers. We value great, well-written documentation and aim to improve it as often as possible. And we're always looking for help with documentation!

19.4.1 Getting the raw documentation

The official documentation is available on Read the Docs. This is the compiled HTML version. However, we edit it as a collection of text files for maximum flexibility. These files live in the top-level `docs/` directory of a RapidSMS release. If you'd like to start contributing to our docs, get the development version of RapidSMS from the source code repository (see *Installing the latest development version*).

19.4.2 Using Sphinx

Before building the documentation, you must have a version of RapidSMS installed. See the *Installing the latest development version* guide for instructions on installing RapidSMS.

We use the Sphinx documentation system (based on docutils). To build the documentation locally, you'll need to install Sphinx:

```
pip install Sphinx
```

Then, building the HTML is easy. Just run make from the `docs` directory:

```
make html
```

(or `make.bat html` on Windows)

To get started contributing, you'll want to read the reStructuredText Primer. After that, you'll want to read about the Sphinx-specific markup that's used to manage metadata, indexing, and cross-references.

19.4.3 Documentation starting points

Typically, documentation changes come in two forms:

- **General improvements:** typo corrections, error fixes and better explanations through clearer writing and more examples.
- **New features:** documentation of features that have been added to the framework since the last release.

If you're interested in helping out, a good starting point is with the documentation label on the GitHub issue tracker.

19.5 RapidSMS core test suite

We expect all new and changed code to include tests for the new or changed behavior. Having as complete a set of tests as possible is important so we can have confidence that when we make a change in one place, we haven't broken something else.

RapidSMS provides several configuration options and entry points for continuous integration. The unit tests can be run in the current python environment or automated to run in several environments and can include coverage reports.

19.5.1 Quickstart

Note: The RapidSMS core test suite includes some tests that depend on django-celery 3.0+. If you install the requirements file listed below, all tests should pass properly.

The easiest way to run the unit tests in a new install is to run the following from the project root:

```
python setup.py develop
pip install -r tests/requirements/dev.txt
python run_tests.py --settings=tests.default
```

The settings found in the /tests/default.py module are intended as the default settings file for running tests. You can tell the test runner what settings file to use with the –settings flag or by setting your `DJANGO_SETTINGS_MODULE` environment variable.

19.5.2 Coverage

To see code coverage while running the tests, you can use the supplied `coverage` settings file:

```
python run_tests.py --settings=tests.coverage
```

19.5.3 PEP 8 Style Guidelines Adherence

RapidSMS adheres to the Python PEP 8 style guidelines for all Python source outside of the `docs/` directory. As such, please check your code against the PEP 8 specification by using the `flake8` linting tool in your RapidSMS directory before submitting patches for review:

```
flake8
```

19.5.4 Testing multiple environments

RapidSMS uses Tox to run the test suite in a variety of environments. You can test all environments together or specific ones:

```
tox                 # all environments
tox -e py27-1.7.X   # only test using Python 2.7 and Django 1.7
```

The available environments are:

- `py26-1.4.X` - Test using Python 2.6 and Django 1.4.X
- `py26-1.5.X` - Test using Python 2.6 and Django 1.5.X
- `py26-1.6.X` - Test using Python 2.6 and Django 1.6.X
- `py27-1.4.X` - Test using Python 2.7 and Django 1.4.x
- `py27-1.5.X` - Test using Python 2.7 and Django 1.5.x
- `py27-1.6.X` - Test using Python 2.7 and Django 1.6.x
- `py27-1.7.X` - Test using Python 2.7 and Django 1.7.x
- `py27-trunk` - Test using Python 2.7 and Django master
- `docs` - Test building docs

- `flake8` - Run flake8 style checker
- `coverage` - Report coverage

You can also add additional environments or change other parts of the configuration in your local copy of the tox.ini by following the tox configuration specification docs.

19.5.5 Using setup.py

Your `DJANGO_SETTINGS_MODULE` must be set in order for the test suite using setup.py. Running the following will install test dependencies and run the unit tests in one step, but without the option of a coverage report:

```
export DJANGO_SETTINGS_MODULE=tests.default
python setup.py test
```

19.6 RapidSMS Release Checklist

This is a checklist for releasing a new version of RapidSMS.

This is intended for someone who has been designated the *Release Manager*, the person responsible for making an official release of RapidSMS.

For a higher-level overview of the whole release cycle, see the *Release Process*.

19.6.1 Git branches

RapidSMS uses the Git Flow process for development. The two branches of concern at release time are:

- **master** - always has the most recently released code. Each release is tagged `vX.X.X`.
- **develop** - contains the code under development for the next release

So technically, what makes a new release is merging `develop` to `master` and tagging it. Of course, we don't want to do that until we're ready.

19.6.2 Required permissions

You'll need the following authorizations to make a release:

Github push to the rapidsms repository

PyPI push updates to the rapidsms package

Read the Docs change the configuration for RapidSMS

If you need any of these authorizations:

- create any of the accounts you don't have
- send an email to rapidsms-dev@googlegroups.com and ask for someone to give you the needed authorizations. Be sure to include your userid for each account.

19.6.3 Release checks

All the following checks should be verified before continuing:

- `master` merged to `develop` to be sure any hotfixes are included
- Version number in `rapidsms/__init__.py` updated
- *Next* version number in `rapidsms/docs/conf.py` updated
- New release labeled as current in `rapidsms/docs/releases` in `index.rst`, `this-release.rst`, and `roadmap.rst`
- Previous release not labeled as current in `rapidsms/docs/releases` in `index.rst`, `prev-release.rst`, and `roadmap.rst`.
- Create a `release-X.X.X` branch (based off `master`) in the RapidSMS project template repository. Make sure to update `README.rst` as well.
- Update project template command line in `intro/install/index.rst` to point to the RapidSMS project template release branch.
- All git issues for this release's milestone have been resolved. (closed or moved to another milestone)
- All tests pass against the `develop` branch. Look for a passing build on Travis of the tip commit on the `develop` branch.
- A distribution tarball can be built with `python setup.py sdist`, it can be installed with pip, has the right version, and works when installed. (SUGGEST HOW TO TEST AN INSTALLED RAPIDSMS)

19.6.4 Release steps

Take these steps to release the new version:

- Make a fresh clone of the repo:

```
git clone git@github.com:rapidsms/rapidsms.git
cd rapidsms
```

- Checkout master:

```
git checkout master
```

- Merge develop into master:

```
git merge develop
```

- Run the tests locally. (This assumes you have tox on your path. Create a new virtualenv and install it if needed.) The tests must pass before proceeding.

```
tox
```

- Create a new tag:

```
git tag -a vX.X.X
```

- Push the merged master branch and tag to github:

```
git push origin master --tags
```

- While Travis is testing the pushed branch, compose a release announcement.

Here's a template that can be used for release announcements. You can copy the summary from the release notes:

Subject: RapidSMS X.X.X Released

I'm excited to announce the release of RapidSMS X.X.X! Here's a quick summary:

- **Major change or feature 1:** *one-line explanation*
- **Major change or feature 2:** *one-line explanation*
- ...

You can find the full list of changes and upgrade guide in the RapidSMS X.X.X Release Notes.

I'd like to give special thanks to Tom, Dick, and Harry for their work on this release. *[EXPAND ON THAT].*

More help is always welcome. If you're interested, you can read the contributing guide.

The next release will be *Y.Y.Y* and will focus on *FILL IN MAJOR GOALS FOR Y.Y.Y.*

As always, if you have any questions or issues, please feel free to post them to this list or ask in the #rapidsms IRC channel on Freenode. Bugs can be reported on Github.

- Verify that Travis tests have passed for the pushed master
- Push the new version to PyPI:

```
python setup.py sdist upload
```

- Add the new version to the tags that Read the Docs should build
- Email the release announcement to rapidsms@googlegroups.com and rapidsms-dev@googlegroups.com

19.6.5 Start Next Release

Back on the `develop` branch, we can now start on the next release:

- Merge `master` to `develop` to make sure we're starting from the same code that's currently released (there might have been merge conflicts or something during the release process).
- Update the version in `rapidsms/__init__.py` and the next version in `rapidsms/docs/conf.py`.
- Start a new release notes document in `rapidsms/docs/release`. Use the previous release's document as a template. Be sure to label it at the top as under development.
- Update `rapidsms/docs/release/index.rst` to mark the next release as under development.
- Create a new Github milestone with the next release number, e.g. "v0.15.0", so that developers can start targeting work for the next release.

Now we can start merging features intended for the next release. Review Git Flow for more about how to use git branches while developing.

CHAPTER 20

The RapidSMS Community

The RapidSMS community is everyone who helps make RapidSMS what it is.

This section of the documentation is about the RapidSMS community, and how it works.

See *Developing RapidSMS* for technical information about making changes to RapidSMS.

When working with the RapidSMS community, it'll help to remember a few things:

- At the other end of each email address is another person.
- Most of the people in the community contribute because they want to help, not because it's their job.

So - be polite, give each other the benefit of the doubt, and try to communicate clearly but be ready to try another approach if your point doesn't seem to be getting across.

20.1 Joining the RapidSMS community

We're always excited to welcome new volunteers to the RapidSMS project. As it keeps growing, we need more people to help others and contribute back to the community. As soon as you learn RapidSMS, you can contribute in many ways:

- Join the rapidsms mailing list and answer questions. This users list, the primary RapidSMS mailing list, is used to ask and answer questions, help work through bugs, and discuss general RapidSMS topics.
- Join the rapidsms-dev mailing list. This list is used to discuss RapidSMS core and active country projects. This ranges from new RapidSMS features to large scale RapidSMS deployments in Zambia. If you're interested in any of this, please join the conversation!
- Join the #rapidsms IRC channel on Freenode and answer questions. By explaining RapidSMS to other users, you're going to learn a lot about the framework yourself. You can use the webchat client for this too.

And, of course, you can help out by working *on* RapidSMS.

- Report bugs in our ticket tracker.
- Follow our *Coding standards and best practices*.

- Open *pull requests* for features and bug fixes against the *develop* branch. We use the Gitflow model for our development. See *Submitting changes back to the project* for more on this.
- Comment on open issues and pull requests. Try the changes yourself and report on how well they work in the issue or pull request. See *Reviewing pull requests* for more on this.
- Try the `develop` branch and report any problems, especially if recent changes break your own application.
- Improve the RapidSMS *documentation* (what you're reading now!).
- Participate on the rapidsms-dev mailing list.

20.2 Submitting changes back to the project

Here's how you can offer your changes back to the RapidSMS project, so others can benefit from them.

(If you're a member of *The RapidSMS core team*, you should still follow this process.)

20.2.1 See if there's support for your change

1. Check for open issues or open a fresh issue in the ticket tracker to start a discussion around a feature idea or a bug. Send a message to the rapidsms-dev mailing list to request feedback.
2. If you're working on a large patch, we highly recommend creating a wiki page under the RapidSMS GitHub account. Use the wiki page to outline the motivation behind the patch and to document decisions made on the rapidsms-dev mailing list.

   Router decoupling and HTTP message processing, Bulk Messaging API, and Scheduling are good examples.

20.2.2 Develop your change

Write the code for your change.

Follow the RapidSMS *Coding standards and best practices*.

20.2.3 Sign the CLA

Before your work can be accepted into the project, you'll have to sign a *Contributor License Agreement*.

20.2.4 Ask for it to be added

To ask the project to merge your changes into the official repository, *Submit a pull request*.

20.2.5 Now what?

What you'd like to happen is for someone with privileges on the RapidSMS repository to merge your changes into the `develop` branch, so that they'll be included in the next RapidSMS release.

Typically before that, there will be one or more rounds of people making comments on your changes or asking questions to try to improve the code.

If you don't hear anything after a few days, it's okay to ask on the `rapidsms-dev` mailing list for someone to look at your pull request. Everyone is busy and sometimes a reminder is helpful.

Just remember that if you want your change included, you'll need to take responsibility to convince people that it's worthwhile. And sometimes perfectly good changes are not included in the project because they don't appear to be useful to enough users. Every time code is added, it's a cost to the project because it has to be maintained from then on.

You can always continue using the change in your fork, even if it's not included in the main project.

20.2.6 Core team members

If you're a member of *The RapidSMS core team*, there are a few small changes to the process.

You can merge your own changes, but if possible, you should first get another core team member to agree with them. Ask them to add a "ship it" comment to the pull request. After that, you can merge your changes.

It's also your prerogative to merge your own changes if no one has commented after a few days. Just keep in mind that whatever changes you make will contribute to your own standing in the community.

20.3 Reviewing pull requests

Everyone is welcome to comment on pull requests. If you're deeply familiar with the RapidSMS code, you might spot a fundamental problem. But if not, you still can look for smaller-scale bugs, typos, style issues, better algorithms, etc.

The *Coding standards and best practices* are a good checklist when reviewing pull requests. All pull requests are expected to comply with our coding standards.

Check out the code from the pull request's branch and try running it locally. Make sure the tests pass on your system; it might be different from the submitter's system. Try other things; even if all the tests pass, there might be things we forgot to write a test for, or couldn't.

See if your own app still works when using the modified RapidSMS. Sometimes different apps use RapidSMS in different ways.

If you are on *The RapidSMS core team*, you're encouraged to review pull requests and if they look acceptable, merge them to the `develop` branch. Or if the submitter is another core team member, add a `ship it!` comment and let them do the merging.

In any case, please try to be polite in your comments, and give the submitter the benefit of the doubt. Before clicking to submit your comment, think about how you would feel if someone made that comment about your code.

One good policy when writing review comments is to always refer to the code, never to the submitter. Remember that what we're reviewing is the code, not the person. For example, instead of saying "you didn't check for None", you might instead say "the method is missing a check for None".

20.4 The RapidSMS core team

The RapidSMS team is fairly loosely organized. The `core team` is basically anyone who has permission to merge into the official repository.

The only real prerequisite to joining the core team is showing that you'd be a useful member of the team. So, start by *contributing*. After you've done this for a while, someone on the core team might decide to give you access. If not, feel free to ask on the `rapidsms-dev` list, where by this time you should be well known anyway.

You can view the current core team on the RapidSMS GitHub project page.

There are some distinguished roles on the team. Note that any person can play multiple roles.

20.4.1 Core Dev

A core dev is a community member who has commit rights to the GitHub repository. They've usually been active for some time, have a track record of submitting quality pull requests and help review outstanding PRs.

20.4.2 Release Manager

A release manager is a member of the core dev team who performs each release when feature development is complete. They collaborate with the *Release Champion* on the *Release process*, provide final QA before each release, and actually perform the final *release process steps* to publish the new release.

20.4.3 Community Coordinator

A community coordinator is a core dev who helps organize the core team to identify future milestones and releases. They are active on the mailing lists and help identify release champions for each new version.

20.4.4 Release Champion

A release champion is a core dev who is excited about the upcoming release, has expertise in the topic and wants to lead development of the proposed features. They will organize tickets, delegate tasks to other willing core devs and perform actual development themselves. Their job is expanded on in the *Release process*.

20.5 Release process

The process for a new RapidSMS release begins when a *Release Champion* is identified for it. The release champion's job is to organize and lead the work to get a new release out. Any of the work identified below could be done by anyone, under the release champion's leadership.

The release champion starts by creating a new Github milestone for the release, if there's not one already. It will be named something like "v0.15.0".

There might already have been changes merged to the `develop` branch since the last release. The release champion should find those closed issues and add the new milestone to them, so the new milestone will contain all the issues that'll be in the new release.

Then the release champion will look for existing issues, or create new ones, that they want to include in the new release. They'll add the new milestone to each of those issues.

The release champion should use the rapidsms-dev mailing list to inform everyone interested in the development of RapidSMS of the progress of the release. In particular, they should solicit suggestions for issues that should be included in the new release, and ask people to try out the `develop` branch as changes are merged.

The release champion and *Release Manager* collaborate when it's time to ship the new release, with the release manager responsible for final QA and the actual *RapidSMS Release Checklist* steps to publish a new release.

CHAPTER 21

Contributed Applications

RapidSMS comes with a number of contributed applications.

21.1 rapidsms.contrib.default

The *default* contrib application allows your project to define a default response to an *IncomingMessage* that is not handled by any other application. The response string is defined in *DEFAULT_RESPONSE*.

21.1.1 Installation

To use *default*, add "rapidsms.contrib.default" to the end of INSTALLED_APPS in your settings file:

```
INSTALLED_APPS = [
    # Your other installed apps
    ...
    "rapidsms.contrib.default"  # must be last
]
```

21.1.2 Usage

The *default* application operates during the *default message processing stage*. It is very important that the router loads this application last, both because all other applications should have the opportunity to handle the message before falling back to this one, and because this application does not prevent the execution of the default stages of the applications that come after it.

This application passes the value of *PROJECT_NAME* to the response string. To include the project name, use %(project_name)s in the response string.

If *DEFAULT_RESPONSE* is None, the *default* application will not send a message.

By default, *DEFAULT_RESPONSE* is defined as:

```
DEFAULT_RESPONSE = "Sorry, %(project_name)s could not understand your message."
```

21.2 rapidsms.contrib.echo

The *echo* contrib application is a collection of two simple handlers that can assist you in remotely debugging your RapidSMS project.

21.2.1 Installation

To use either of *EchoHandler* or *PingHandler*, you must add `"rapidsms.contrib.handlers"` to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.handlers",
    ...
]
```

Then add the handler classes you want to use to *RAPIDSMS_HANDLERS*:

```
RAPIDSMS_HANDLERS = [
    ...
    "rapidsms.contrib.echo.handlers.echo.EchoHandler",  # if you want EchoHandler
    "rapidsms.contrib.echo.handlers.ping.PingHandler",  # if you want PingHandler
    ...
]
```

See the handlers docs for more information about how handlers are loaded and called.

21.2.2 Usage

EchoHandler

EchoHandler is an extension of *KeywordHandler* which handles any message prefixed by "echo" by responding with the remainder of the text. This handler is useful for remotely testing internationalization.

For example:

```
> echo
< To echo some text, send: ECHO <ANYTHING>
> echo hello
< hello
```

PingHandler

PingHandler is an extension of *BaseHandler*. It handles messages with the (precise) text "ping" by responding with "pong". Unlike many handlers, this one is case-sensitive and does not allow extra whitespace. This handler is useful for remotely checking that the router is alive.

For example:

```
> ping
< pong
```

21.3 rapidsms.contrib.handlers

The *handlers* contrib application provides three classes- *BaseHandler*, *KeywordHandler*, and *PatternHandler*- which can be extended to help you create RapidSMS applications quickly.

21.3.1 Installation

To define and use handlers for your RapidSMS project, you will need to add `"rapidsms.contrib.handlers"` to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.handlers",
    ...
]
```

Then you'll also need to set *RAPIDSMS_HANDLERS*. The application will load the handler classes listed in *RAPIDSMS_HANDLERS*, as described in *Handler Discovery*.

```
RAPIDSMS_HANDLERS = [
    "rapidsms.contrib.handlers.KeywordHandler",
    "rapidsms.contrib.handlers.PatternHandler",
]
```

21.3.2 Usage

KeywordHandler

Many RapidSMS applications operate based on whether a message begins with a specific keyword. By subclassing *KeywordHandler*, you can easily create a simple, keyword-based application:

```
from rapidsms.contrib.handlers import KeywordHandler

class LightHandler(KeywordHandler):
    keyword = "light"

    def help(self):
        self.respond("Send LIGHT ON or LIGHT OFF.")

    def handle(self, text):
        if text.upper() == "ON":
            self.respond("The light is now turned on.")

        elif text.upper() == "OFF":
            self.respond("Thanks for turning off the light!")

        else:
            self.help()
```

Your handler must define three things: *keyword*, *help()*, and *handle(text)*. When a message is received that begins with the keyword (case insensitive; leading whitespace is allowed), the remaining text is passed to the *handle* method of the class. If no additional non-whitespace text is included with the message, *help* is called instead. For example:

```
> light
< Send LIGHT ON or LIGHT OFF.
> light on
< The light is now turned on.
> light off
< Thanks for turning off the light!
> light something else
< Send LIGHT ON or LIGHT OFF.
```

The handler also treats `,`, `:`, and `;` after the keyword the same as whitespace. For example:

```
> light
< Send LIGHT ON or LIGHT OFF.
> light:on
< The light is now turned on.
> light, off
< Thanks for turning off the light!
> light :,; on
< The light is now turned on.
```

Tip: Technically speaking, the incoming message text is compared to a regular expression pattern.

The most common use case is to look for a single exact-match keyword. However, one could also match multiple keywords, for example `keyword = "register|reg|join"`.

However, due to how we build the final regular expression, capturing matches using grouping in the keyword regular expression won't work. If you need that, use the *PatternHandler*.

All non-matching messages are silently ignored to allow other applications and handlers to catch them.

For example implementations of *KeywordHandler*, see

- rapidsms.contrib.echo.handlers.echo.EchoHandler
- rapidsms.contrib.registration.handlers.register.RegistrationHandler
- rapidsms.contrib.registration.handlers.language.LanguageHandler

Here's documentation from the KeywordHandler class:

class `rapidsms.contrib.handlers.`**`KeywordHandler`**

This handler type can be subclassed to create simple keyword-based handlers. When a message is received, it is checked against the mandatory `keyword` attribute (a regular expression) for a prefix match. For example:

```
>>> class AbcHandler(KeywordHandler):
...     keyword = "abc"
...
...     def help(self):
...         self.respond("Here is some help.")
...
...     def handle(self, text):
...         self.respond("You said: %s." % text)
```

If the keyword is matched and followed by some text, the `handle` method is called:

```
>>> AbcHandler.test("abc waffles")
['You said: waffles.']
```

If *just* the keyword is matched, the `help` method is called:

```
>>> AbcHandler.test("abc")
['Here is some help.']
```

All other messages are silently ignored (as usual), to allow other apps or handlers to catch them.

`handle`(*text*)

Called when the keyword matches and text follows

Parameters **`text`** – The text that follows the keyword. Any whitespace between the keyword and the text is not included.

help()
Called when the keyword matches but no text follows

keyword = None
A string specifying a regular expression matched against the beginning of the message. Not case sensitive.

PatternHandler

Note: Pattern-based handlers can work well for prototyping and simple use cases. For more complex parsing and message handling, we recommend writing a RapidSMS application with a custom *handle phase*.

The *PatternHandler* class can be subclassed to create applications which respond to a message when a specific pattern is matched:

```
from rapidsms.contrib.handlers import PatternHandler

class SumHandler(PatternHandler):
    pattern = r"^(\d+) plus (\d+)$"

    def handle(self, a, b):
        a, b = int(a), int(b)
        total = a + b
        self.respond("%d + %d = %d" % (a, b, total))
```

Your handler must define *pattern* and *handle(*args)*. The pattern is case-insensitive, but must otherwise be matched precisely as written (for example, the handler pattern written above would not accept leading or trailing whitespace, but the pattern `r"^(\d+) plus (\d+)\s*$"` would allow trailing whitespace). When the pattern is matched, the *handle* method is called with the captures as arguments. As an example, the above handler could create the following conversation:

```
> 1 plus 2
< 1 + 2 = 3
```

Like *KeywordHandler*, each *PatternHandler* silently ignores all non-matching messages to allow other handlers and applications to catch them.

Here's documentation from the PatternHandler class:

class rapidsms.contrib.handlers.**PatternHandler**
This handler type can be subclassed to create simple pattern-based handlers. This isn't usually a good idea – it's cumbersome to write patterns with enough flexibility to be used in the real world – but it's very handy for prototyping, and can easily be upgraded later.

When a message is received, it is matched against the mandatory `pattern` attribute (a regular expression). If the pattern is matched, the `handle` method is called with the captures as arguments. For example:

```
>>> class SumHandler(PatternHandler):
...     pattern = r'^(\d+) plus (\d+)$'
...
...     def handle(self, a, b):
...         a, b = int(a), int(b)
...         total = a + b
...
...         self.respond(
...             "%d+%d = %d" %
...             (a, b, total))
```

```
>>> SumHandler.test("1 plus 2")
['1+2 = 3']
```

Note that the pattern must be matched *precisely* (excepting case sensitivity). For example, this would not work because of the trailing whitespace:

```
>>> SumHandler.test("1 plus 2 ")
False
```

All non-matching messages are silently ignored, to allow other apps or handlers to catch them.

handle(**args*)
Called when the message matches the pattern. Any matching groups are passed to it.

Parameters **args** – The matching groups from the regular expression.

pattern = None
A string specifying a regular expression that should match the message. Not case sensitive.

BaseHandler

All handlers, including the *KeywordHandler* and *PatternHandler*, are derived from the *BaseHandler* class. When extending from *BaseHandler*, one must always override the class method *dispatch*, which should return `True` when it handles a message.

All instances of *BaseHandler* have access to *self.msg* and *self.router*, as well as the methods *self.respond* and *self.respond_error* (which respond to the instance's message).

BaseHandler also defines the class method *test*, which creates a simple environment for testing a handler's response to a specific message text. If the handler ignores the message then `False` is returned. Otherwise a list containing the *text* property of each *OutgoingMessage* response, in the order which they were sent, is returned. (Note: the list may be empty.) For example:

```
>>> from rapidsms.contrib.echo.handlers.echo import EchoHandler
>>> EchoHandler.test("not applicable")
False
>>> EchoHandler.test("echo hello!")
["hello!"]
```

For an example implementation of a *BaseHandler*, see rapidsms.contrib.echo.handlers.ping.PingHandler.

21.3.3 Calling Handlers

When a message is received, the *handlers* application calls *dispatch* on each of the handlers it loaded during *handlers discovery*.

The first handler to accept the message will block all others. The order in which the handlers are called is not guaranteed, so each handler should be as conservative as possible when choosing to respond to a message.

21.3.4 Handler Discovery

Please, see the release notes Handlers may be any new-style Python class which extends from one of the core handler classes, e.g. *BaseHandler*, *PatternHandler*, *KeywordHandler*, etc.

The Python package names of the handler classes to be loaded should be listed in *RAPIDSMS_HANDLERS*.

Example:

```
RAPIDSMS_HANDLERS = [
    "rapidsms.contrib.handlers.KeywordHandler",
    "rapidsms.contrib.handlers.PatternHandler",
]
```

Warning: The behavior described in the rest of this section is the old, deprecated behavior. If *RAPIDSMS_HANDLERS* is set, the older settings are ignored.

Handlers may be defined in the *handlers* subdirectory of any Django app listed in INSTALLED_APPS. Each file in the *handlers* subdirectory is expected to contain exactly one new-style Python class which extends from one of the core handler classes.

Handler discovery, which occurs when the *handlers* application is loaded, can be configured using the following project settings:

- *RAPIDSMS_HANDLERS_EXCLUDE_APPS* - The application will not load handlers from any Django app included in this list.
- *INSTALLED_HANDLERS* - If this list is not None, the application will load only handlers in modules that are included in this list.
- *EXCLUDED_HANDLERS* - The application will not load any handler in a module that is included in this list.

Note: Prefix matching is used to determine which handlers are described in *INSTALLED_HANDLERS* and *EXCLUDED_HANDLERS*. The module name of each handler is compared to each value in these settings to see if it starts with the value. For example, consider the *rapidsms.contrib.echo* application which contains the *echo* handler and the *ping* handler:

- "rapidsms.contrib.echo.handlers.echo" would match only *EchoHandler*,
- "rapidsms.contrib.echo" would match both *EchoHandler* and *PingHandler*,
- "rapidsms.contrib" would match all handlers in any RapidSMS contrib application, including both in *rapidsms.contrib.echo*.

21.4 rapidsms.contrib.httptester

The *httptester* contrib application allows sending fake messages to RapidSMS and seeing how RapidSMS responds.

21.4.1 Installation

To define and use Message Tester for your RapidSMS project, you will need to:

1. Add "rapidsms.contrib.httptester" and "rapidsms.backends.database" to INSTALLED_APPS in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.backends.database",
    "rapidsms.contrib.httptester",
    ...
]
```

2. Add *httptester* URLs to your urlconf:

```
urlpatterns = patterns("",
    ...
    (r"^httptester/", include("rapidsms.contrib.httptester.urls")),
    ...
)
```

3. Add the Database backend to *INSTALLED_BACKENDS* with the name "message_tester":

```
    INSTALLED_BACKENDS = {
    ...
    "message_tester": {
        "ENGINE": "rapidsms.backends.database.DatabaseBackend",
    },
    ...
}
```

4. Create database tables for the DB backend models:

```
$ python manage.py syncdb
```

5. Add a link to the Message Tester view from your rapidsms/_nav_bar.html template:

```
{% load url from future %}
<li><a href="{% "httptester-index" %}">Message Tester</a></li>
```

21.4.2 Usage

With Message Tester installed, there will be a *Message Tester* tab in the RapidSMS web page header. Click on that tab to bring up Message Tester.

Most of the controls for the Message Tester are in the left-side column.

The phone number field contains the phone number which will be used as the source number when you send test messages. A random number will have been filled in for you, but you can change it to anything you want.

You can send a single message by typing the message in the *Single Message* field and clicking *Send*. Or you can send multiple messages by putting each message on one line of a text file, selecting that text file with the *Choose File* button, and clicking *Send*.

The Log table on the right side of the page will show messages you send, and any messages that RapidSMS replies with. For messages that you send, the left column will show the phone number the message came from, and a double arrow pointing right, with the text of the message in the right column. For messages that RapidSMS sends, the left column will show the phone number the message was sent to, and a double arrow pointing left, with the text of the message again in the right column.

The Log table will always show the most recent messages. If there are more than will fit on a page, you can use the paging controls at the bottom of the table to page back through the messages.

You can clear the log of messages for the current phone number by selecting the *Clear* checkbox and clicking *Send*, or the entire log by selecting the *Clear all* checkbox and clicking *Send*.

21.5 rapidsms.contrib.locations

Warning: rapidsms.contrib.locations will be removed from contrib in the next version of RapidSMS.

Locations allows you to easily map custom locations and points in your i RapidSMS project.

21.5.1 Installation

1. The *locations* contrib application depends on djtables to display data. You can install *djtables* using pip:

```
pip install djtables
```

2. Add `"rapidsms.contrib.locations"` and `"djtables"` (if not already present) to `INSTALLED_APPS` in your settings file:

```
    INSTALLED_APPS = [
        ...
        "rapidsms.contrib.locations",
        "djtables",
        ...
    ]
```

3. Add *locations* URLs to your urlconf:

```
    urlpatterns = patterns("",
        ...
        (r"^locations/", include("rapidsms.contrib.locations.urls")),
        ...
    )
```

4. Create database tables for the *locations* models:

```
$ python manage.py syncdb
```

5. If wanted, add a navigation item to your `rapidsms/_nav_bar.html` template:

```
{% load url from future %}
<li><a href="{% url "locations" %}">Map</a></li>
```

21.5.2 Usage

Locations will auto-generate a map and editing interface for any models that inherit from *rapidsms.contrib.locations.models.Location*. For example, say you had an application called *cities* with a *City* model:

```
# example file: cities/models.py

from django.db import models
from rapidsms.contrib.locations.models import Location

class City(Location):
    name = models.CharField(max_length=100)

    class Meta(object):
        app_label = "cities"
        verbose_name_plural = "cities"
```

To use Locations, you'd add *cities* to your installed apps:

```
INSTALLED_APPS = [
    ...
```

```
    "cities",
    ...
]
```

Create the necessary database tables:

```
$ python manage.py syncdb
```

Now visit the Map tab in your browser to see the *City* model.

21.6 rapidsms.contrib.messagelog

The *messagelog* contrib application maintains a database record of all messages sent and received by RapidSMS.

21.6.1 Installation

1. Add `"rapidsms.contrib.messagelog"` to `INSTALLED_APPS` in your settings file:

```
    INSTALLED_APPS = [
        ...
        "rapidsms.contrib.messagelog",
        ...
    ]
```

2. Add *messagelog* URLs to your urlconf:

```
    urlpatterns = patterns("",
        ...
        (r"^messagelog/", include("rapidsms.contrib.messagelog.urls")),
        ...
    )
```

3. Create database tables for the *messagelog* models:

```
$ python manage.py syncdb
```

4. Optionally, add a link to the message log view from your `rapidsms/_nav_bar.html` template:

```
{% load url from future %}
<li><a href="{% url "message_log" %}">Message Log</a></li>
```

21.6.2 Usage

messagelog defines the *Message* database model, which stores key information about an *IncomingMessage* or *OutgoingMessage*:

connection The RapidSMS *Connection* to which the message was sent.

contact The RapidSMS *Contact* associated with the connection.

date When the message was sent.

text The text of the message.

direction *Message.INCOMING* or *Message.OUTGOING*.

Upon *parsing an IncomingMessage*, *messagelog* creates a *Message* object and sets the *logger_msg* property of the *IncomingMessage* to be the *Message* object.

Upon *processing an OutgoingMessage*, *messagelog* creates a *Message* object and sets the *logger_msg* property of the *OutgoingMessage* to be the *Message* object.

You can navigate to the *message_log* view to browse the full list of stored messages.

21.7 rapidsms.contrib.messaging

The *messaging* contrib application allows you to send messages to one or more recipients through a web interface.

21.7.1 Installation

1. The *messaging* contrib application depends on django-selectable to create a recipient multi-selector with auto-complete on the front-end view. You can install *django-selectable* using pip:

```
pip install django-selectable
```

2. Add `"rapidsms.contrib.messaging"` and `"selectable"` (if not already present) to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.messaging",
    "selectable",
    ...
]
```

3. Add URLs for *messaging* and *selectable* to your urlconf:

```
urlpatterns = ("",
    ...
    (r"^messaging/", include("rapidsms.contrib.messaging.urls")),
    (r"^selectable/", include("selectable.urls")),
    ...
)
```

21.7.2 Usage

The messaging front-end view displays a form through which you can write a text message and select its recipients. The recipient selector uses autocomplete to search through all RapidSMS connections. You may select any number of recipients to receive the message.

When sending a message, the messaging application calls *`rapidsms.router.send()`* with the message text and *recipient.default_connection* for each recipient. If an error occurs, the message will not be sent to further recipients but it may have already been sent to earlier recipients. The order in which messages will be sent is not guaranteed.

21.8 rapidsms.contrib.registration

The *registration* app provides a nice interface for creating, updating, and deleting RapidSMS contacts, both on the web and via SMS messages. It is deliberately minimal, and outside of the core, so other apps can extend or replace it where necessary.

21.8.1 Installation

1. *registration* depends on handlers, so first install *handlers* by adding `"rapidsms.contrib.handers"` to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.handlers",
    ...
]
```

2. Add `"rapidsms.contrib.registration"` to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "rapidsms.contrib.handlers",
    "rapidsms.contrib.registration",
    ...
]
```

3. Add URLs for *registration* to your urlconf:

```
urlpatterns = ("",
    ...
    (r"^registration/", include("rapidsms.contrib.urls")),
    ...
)
```

4. Add *registration*'s handlers to *RAPIDSMS_HANDLERS*:

```
RAPIDSMS_HANDLERS = [
    ...,
    "rapidsms.contrib.registration.handlers.language.LanguageHandler",
    "rapidsms.contrib.registration.handlers.register.RegisterHandler",
]
```

5. (Optional) add *registration* link to the nav bar:

```
{% load url from future %}
<li><a href="{% url 'registration' %}">Registration</a></li>
```

21.8.2 Usage

The *registration* app provides both web and SMS interfaces.

Web

At the left of each page is a set of links:

- List Contacts
- Add Contact
- Bulk Add Contacts

List Contacts is the front page. It displays a table with the contacts. You can click on a contact row to edit that contact. You can edit the contact's name, language, etc., and also edit their connections near the bottom. A blank connection form is at the bottom; add a new connection by filling in the blank form's Backend and Identity fields and saving. Each

existing connection has a Delete checkbox; delete a connection by checking its checkbox and saving. You can delete a contact by clicking the Delete Contact button at the bottom.

Add Contact goes to a blank form for adding a new contact. It works just like the page for editing a contact.

Bulk Add Contacts allows creating many contacts at once by uploading a .CSV file with the data. There's help on the page showing the format that the file should have.

SMS Messages

Users can use SMS messages to register themselves or change their preferred language through the *register* app.

REGISTER

Users can create a contact for themselves along with a connection representing their backend and identity by sending a text message of the form:

```
REGISTER <name>
```

They can also use `REG` or `JOIN` as synonyms for `REGISTER`.

LANGUAGE

After they have registered, users can choose their preferred language by sending a text message of the form:

```
LANGUAGE <code>
```

They can also use `LANG` as a synonym for `LANGUAGE`.

The `<code>` should be the international code for the language, e.g. `pt-BR` for Brazilian Portuguese or `de` for German.

CHAPTER 22

Release Notes

Release notes for the official RapidSMS releases. Each release note will tell you what's new in each version, describe any backwards-incompatible changes made, and provide, if needed, any upgrade instructions. Please see the RapidSMS Roadmap for details on upcoming releases.

22.1 RapidSMS 0.22.0 release notes (under development)

Welcome to RapidSMS 0.22.0! These release notes cover the new features in 0.22.0 as well as some *Backwards-incompatible changes in RapidSMS 0.22.0* you'll want to be aware of when upgrading from RapidSMS 0.21.0.

22.1.1 What's New

-

22.1.2 Backwards-incompatible changes in RapidSMS 0.22.0

-

22.2 RapidSMS 0.21.0 release notes (current release)

Welcome to RapidSMS 0.21.0! These release notes cover the new features in 0.21.0 as well as some *Backwards-incompatible changes in RapidSMS 0.21.0* you'll want to be aware of when upgrading from RapidSMS 0.20.0.

22.2.1 What's New

- Fix 'missing migration' issues #459 and #460
- (Fixed #465) Allow backends to specify failed identities back to router.
- (Fixed #439) Allow newlines in KeywordHandler. [Thanks @knightsamar!]

22.2.2 Backwards-incompatible changes in RapidSMS 0.21.0

- The fix for #460 involves dropping a table which was inadvertently created by a complicated bug. The table is the `BackendMessage` which *should not* have valuable data in it, and there is a migration included which should copy any data to the proper table, but we recommend that you back up your database before making this upgrade, especially if you have valuable data in the BackendMessage table.
- The fix for #465 slightly alters the API between Backends and Routers. Previously, there was only one way for the Backend to communicate information about the status of outgoing messages back to the Router. It could raise an exception. The problem is that it's possible that the Backend was responsible for sending a message to multiple identities and that some of those messages could be successful and some could fail. Routers that automatically retry failures (only the `DatabaseRouter`, currently) had no way of knowing which identities failed, so if they just retried all failures, then some identities would receive multiple messages. The fix for #465 allows Backends to add a `failed_identities` parameter to the exception, which allows the Router to retry failures more intelligently.

  What should you change? Changes are optional. Things should work as they did before without changes, but the following changes may improve reliability:

  - Router authors: Look at the `DatabaseRouter` for a working example of how to deal with `failed_identities`. Basically, it calls `Router.send_to_backend`, catching any Exception and looking for a `failed_identities` parameter. It then can retry only those identities.
  - Backend authors: `Backend.send` should try sending messages to all identities, keeping a list of failures and returning that list in the exception. If there are no failures, then it should return None. Making this change will allow specific Routers to behave more intelligently. See the new documentation for `Backend.send`, which is copied below:

    class `rapidsms.backends.base.`**`BackendBase`**(*router*, *name*, ***kwargs*)

    Base class for outbound backend functionality.

    `send`(*id_*, *text*, *identities*, *context=None*)

    Backend sending logic. The router will call this method for each outbound message. This method must be overridden by sub-classes. Backends typically initiate HTTP requests from within this method.

    If multiple `identities` are provided, the message is intended for all recipients.

    Any exceptions raised here will be captured and logged by the router. If messages to some identities failed while others succeeded, you can provide that information back to the router by adding a list of the identities which failed in a `failed_identities` parameter on the exception. If you do provide that parameter, then the router should assume that all identities *not* listed in `failed_identities` were successfully sent.

    Example

```
def send(self, id_, text, identities, context):
    failures = []
    for identity in identities:
        result = send_my_message(identity, text, context)
        if result == 'failed':
            failures.append(identity)
    if failures:
        msg = '%d messages failed.' % len(failures)
        raise MessageSendingError(msg, failed_identities=failures)
```

    Parameters

    * **`id_`** – Message ID
    * **`text`** – Message text
    * **`identities`** – List of identities

* **context** – Optional dictionary with extra context provided by router to backend

22.3 RapidSMS 0.20.0 release notes

Welcome to RapidSMS 0.20.0! These release notes cover the new features in 0.20.0 as well as some *Backwards-incompatible changes in RapidSMS 0.20.0* you'll want to be aware of when upgrading from RapidSMS 0.19.0.

22.3.1 What's New

- Fix link in contributing guide. [Thanks @kevgathuku!]
- Python 3 and Django 1.8 support!

22.3.2 Backwards-incompatible changes in RapidSMS 0.20.0

- This release drops support for Django 1.4 and 1.5. The next release will likely drop support for Django 1.6, so we encourage users to start upgrading their installations.

22.4 RapidSMS 0.19.0 release notes

Welcome to RapidSMS 0.19.0! These release notes cover the new features in 0.19.0 as well as some *Backwards-incompatible changes in RapidSMS 0.19.0* you'll want to be aware of when upgrading from RapidSMS 0.18.0.

22.4.1 What's New

- Minor typos and fixes.
- Allow from address configuration with Vumi
- Relax requirements. [Thanks @thedrow!]

22.4.2 Backwards-incompatible changes in RapidSMS 0.19.0

- None

22.5 RapidSMS 0.18.0 release notes

Welcome to RapidSMS 0.18.0! These release notes cover the new features in 0.18.0 as well as some *Backwards-incompatible changes in RapidSMS 0.18.0* you'll want to be aware of when upgrading from RapidSMS 0.17.0.

22.5.1 What's New

- Multiple small documentation fixes [Thanks @jthurner!]
- Minor fix to tutorial
- Remove uses of the deprecated `msg.connection` attribute

- Allow editing of messages in Messagelog admin.
- Change BooleanField default to False [Thanks @shanx!]
- Full Flake8 compliance
- Django 1.7 support [Thanks to multiple contributors who provided PRs including @dpoirier, @johanneswilm, @thedrow, and others.]
- Special thanks to Caktus Group for allowing and encouraging @vkurup to work on the Django 1.7 support during their quarterly ShipIt days!
- Added Coveralls support.

22.5.2 Backwards-incompatible changes in RapidSMS 0.18.0

- Since Django 1.7 has it's own migration framework, South is now deprecated. If you are still using Django < 1.7, you will need to upgrade South to version > 1.0, otherwise it will get confused by migration files which are meant for Django 1.7, not for South. (The `migrations` directories now contain Django 1.7 migrations. South migrations have been moved to a `south_migrations` directory, but older versions of South will not look in that directory.)

22.6 RapidSMS 0.17.0 release notes

Welcome to RapidSMS 0.17.0! These release notes cover the new features in 0.17.0.

22.6.1 What's New

- Fixed link to rapidsms-generic in Community Apps [Thanks @jthurner!]
- Include generated UUID in payload to Vumi [Thanks @vkurup!]
- Fixed router documentation to match returning False in *Outgoing Message Processing* will hault futher processing [Thanks @vkurup!]
- Raise exceptions within `CeleryRouter` background tasks so they can be logged appropriately [Thanks @tobiasmcnulty!]

22.7 RapidSMS 0.16.0 release notes

Welcome to RapidSMS 0.16.0! These release notes cover the new features in 0.16.0 as well as some *Backwards-incompatible changes in RapidSMS 0.16.0* you'll want to be aware of when upgrading from RapidSMS 0.15.0.

22.7.1 What's New

- Updated tox version to fix Travis CI builds
- Tox runs tests against Django 1.4, 1.5, and 1.6
- Allow 160 chars in messages in contrib.messages.

- (Fixed #422) The vumi backend no longer raises errors on empty incoming messages. [Thanks @vkurup!]
- (Fixed #412) The *DatabaseRouter* now commits all transactions before adding background tasks to the queue. [Thanks @dodumosu!]
- (Fixed #409) The *DatabaseRouter* will now properly group batched transmissions across multiple backends. [Thanks @takinbo!]
- (Fixed #411) base.html now uses the Django 1.4+ `static` template tag. [Thanks @wilsonkichoi!]
- Fixed a few typos in Part 1 of the Tutorial.
- Updated `django-celery` in `tests/requirements/dev.txt` to latest version (3.0.23) so `pytz` installs correctly during testing and building the documentation.

22.7.2 Backwards-incompatible changes in RapidSMS 0.16.0

- Style files removed from base: icons.css, modules.css, splits.css, and tables.css. These are no longer used in the new bootstrap-based pages, so they've been removed from the RapidSMS base style files and are no longer loaded from the base template.

  The old `locations` contrib app does still use them, so they've been moved into that app and that app's base template changed to load them from there.

  If you have an app that's still using any of these styles, and aren't ready to convert your app to the new, bootstrap-based look, it's probably simplest for you to copy them from `locations` into your app and load them from your app's base template.

22.8 RapidSMS 0.15.0 release notes

Welcome to RapidSMS 0.15.0! These release notes cover the new features in 0.15.0.

22.8.1 What's New

- **The major changes for 0.15.0 include:**
  - Added a four-part *RapidSMS Tutorial*.
  - Created a *The RapidSMS Community* section in the docs that provides an overview of the community, including the core team.
  - Added a *Developing RapidSMS* section that outlines how to contribute to the RapidSMS codebase.
- *Messages to Users* are displayed at the top of the page if you're using the RapidSMS base template.
- *RAPIDSMS_HANDLERS* is a simpler way to control which handlers are loaded. *INSTALLED_HANDLERS*, *EXCLUDED_HANDLERS*, and *RAPIDSMS_HANDLERS_EXCLUDE_APPS* are deprecated, and `INSTALLED_APPS` are no longer searched automatically for handlers. For more information see *Handler Discovery*.
- Added example of *Submitting changes back to the project*.
- Test class documentation now includes inheritance references. See *rapidsms.tests.harness.RapidTest* for an example.
- Expanded documentation on *Contacts*.
- Added a documentation stub for *packaging reusable apps*.

- Updated README with latest Python dependencies.

22.9 RapidSMS 0.14.0 release notes

Welcome to RapidSMS 0.14.0! These release notes cover the new features in 0.14.0 as well as some *Backwards-incompatible changes in RapidSMS 0.14.0* you'll want to be aware of when upgrading from RapidSMS 0.13.0.

22.9.1 What's New

- New section in the documentation providing advice on *Provisioning Servers & Deploying Your Project*.
- Added navigation links (next, previous, tables of contents) to the documentation and improved the organization a bit.
- Documented the bundled version of Twitter Bootstrap.
- Updated Vumi docs to point to `develop` branch now that the RapidSMS changes have landed.

22.9.2 Backwards-incompatible changes in RapidSMS 0.14.0

- All views built-in to RapidSMS, including contrib apps, now require login.
- Removed *bin/rapidsms-admin.py*. It relied on the RapidSMS *startproject* management command which had already been removed. For the recommended way to start a new RapidSMS project, see *Installing the RapidSMS project template*.

22.10 RapidSMS 0.13.0 release notes

Welcome to RapidSMS 0.13.0! These release notes cover the new features 0.13.0 as well as some *Backwards-incompatible changes in RapidSMS 0.13.0* you'll want to be aware of when upgrading from RapidSMS 0.12.0.

22.10.1 What's New

- **Added support for sending bulk messages. This involved the following changes:**
  - Modified *`MessageBase`* to accept multiple connections.
  - Updated *`rapidsms.router.send()`* to pass multiple connections (within a message object) to the Router. The Messaging API already supported multiple connections, so the API did not change.
  - Updated *`rapidsms.backends.base.BackendBase.send()`* signature to always expect multiple connections and changed the signature to accept `text` and `identities`, rather than a message object. Child classes must now override this function.
  - Updated *`BlockingRouter`* to inspect outbound message connections, group by backend, and call `backend.send` for each one.
- Added the database-powered *`DatabaseRouter`*. The *`DatabaseRouter`* stores all messages in the database and keeps track of received/sent state of each message. This is useful for monitoring the sending of large message volumes.
- Added a backend for Vumi.

- Cleaned up admin for *Connection* to load faster with database JOINs.
- Added *delivery report* functionality for Kannel.
- Combined the BaseRouter and BlockingRouter together. Now all base routing functionality is handled by *BlockingRouter*.
- In order to better support customization of routers, messages now pass through receive_incoming and process_incoming methods. This additional layer allows the *CeleryRouter* and *DatabaseRouter* to more easily customize message processing.
- The Messaging API now asks the router to instantiate incoming and outgoing messages via the new_incoming_message and new_outgoing_message methods. This allows the router to better customize the message classes.
- Reworked the *CeleryRouter* to pass identifiers, rather than instantiated objects, to background tasks. This avoids tasks possibly using outdated data when unpickling task arguments.
- Removed rapidsms.tests.harness.setting.
- Added requests as a dependency. Vumi and Kannel now use requests to construct HTTP requests. You'll need to run pip install requests or add requests to your requirements file.
- Added initial migrations for South support. South handles making incremental changes to database tables as Django models change. To start using:
  - Convert your apps to South
  - Upgrade to RapidSMS 0.13.0
  - Bring database up to the current model definitions:

```
python manage.py syncdb --all
```

  - Tell South that everything is up to date:

```
python manage.py migrate --fake
```

  Once South is set up, future upgrades of RapidSMS should just require manage.py migrate to update your database tables.
- Deprecated rapidsms.log.mixin.LoggerMixin. Please update all logging to use the standard logging module. That includes code that might use LoggerMixin indirectly by extending AppBase, BackendBase, or BaseHandler. Search for code like self.error(...) or self.debug(...) and change to logger.error(...) or logger.debug(...) after creating a logger object as above. You'll want most logging to look like this:

```
import logging
logger = logging.getLogger(__name__)
logger.info(...)
logger.debug(...)
```

- The HTTPTester contrib app has been changed to use the new Database backend instead of its own backend. The HTTPTester backend has been removed. Please remove "rapidsms.contrib.httptester.backend" from INSTALLED_BACKENDS and review the HTTPTester configuration.
- Added created_on and modified_on fields to the Contact and Connection models. On initial migration, any existing records will have those fields set to the current time.

22.10.2 Backwards-incompatible changes in RapidSMS 0.13.0

In the goal of improving the RapidSMS core, we have made a number of backwards- incompatible changes.

Backend configuration must point to a class

In previous versions of RapidSMS, you would define backends like so:

```
INSTALLED_BACKENDS = {
    "kannel-fake-smsc" : {
        "ENGINE":  "rapidsms.backends.kannel",
    }
}
```

Now, backends must specify the name of the class:

```
INSTALLED_BACKENDS = {
    "kannel-fake-smsc" : {
        "ENGINE":  "rapidsms.backends.kannel.KannelBackend",
    }
}
```

This change was made to be more explicit. This also simplifies the importing architecture.

Changed `Backend.send` signature

All exisitng backends must be updated to use the new signature. The router used to pass just a message object to *`BackendBase.send`*. The signature has been updated to accept an `id_`, `text`, list of `identities`, and a `context` dictionary. All backends will need to be updated to use this signature. Please see *`BackendBase.send`* for more details.

Removed start/stop methods

We removed the left over `start` and `stop` methods for the router, backends, and apps. These were important for the legacy, threaded router, but are no longer necessary with new-routing. If your apps and backends use these methods, you'll need to move the functionality to `__init__`.

Removed Message translation functionality

Now that Message objects can contain multiple connections, the internal translation bits needed to change. Messages can be sent to connections that specify different default languages. We removed all translation functionality from the Message objects and require the developer to handle it explicitly.

The internationalization documentation has been updated.

Changed HTTPTester to use Database backend

The HTTPTester contrib app has been changed to use the new Database backend instead of its own backend. The HTTPTester backend has been removed. Please remove `"rapidsms.contrib.httptester.backend"` from `INSTALLED_BACKENDS` and review the HTTPTester configuration.

22.11 RapidSMS 0.12.0 release notes

Welcome to RapidSMS 0.12.0! These release notes cover the new features in 0.12.0 as well as some *backwards incompatible changes in 0.12.0* you'll want to be aware of when upgrading from RapidSMS 0.11.0. Most RapidSMS sites and applications will require some changes when upgrading to RapidSMS 0.12.0.

They key changes in 0.12.0 are:

- **Twitter Bootstrap**: The RapidSMS pages now use Twitter Bootstrap v2.2.2 for a more up-to-date and easily extensible appearance. See Front End for more information.
- **Contrib app updates:** Most of the contrib apps have been updated to use more up-to-date Django practices and to add documentation and tests. A few obsolete apps have been *removed*.
- **Supporting Django 1.4+:** RapidSMS is no longer compatible with any version of Django prior to 1.4. See the Django 1.4 release notes for more information.
- **Removed RAPIDSMS_TABS:** Top level navigation is now managed with an *inclusion template*.
- **Officially adopted Celery**: Following the inclusion of `CeleryRouter`, we've removed `rapidsms.contrib.scheduler` in favor of using Celery directly with RapidSMS. See Using Celery for Scheduling Tasks for more information.

22.11.1 Updating to RapidSMS 0.12.0

You can follow these basic guidelines when upgrading from RapidSMS 0.11.0:

- Upgrade to at least *Django 1.4*
- If upgrading to *Django 1.5*:
- Verify *urls.py* files have been updated
- Verify *url template tags* in templates have been updated
- Set up `rapidsms/_nav_bar.html` to replace the *RAPIDSMS_TABS* setting
- Install `django_tables2` and `django-selectable` via pip or add to your requirements file. See *new dependencies* for more information.
- Add `django_tables2` and `selectable` to `INSTALLED_APPS`.
- Remove references to *removed apps*.

22.11.2 Backwards-incompatible changes in RapidSMS 0.12.0

In the goal of improving the RapidSMS core, we have made a number of backwards- incompatible changes.

Change to Twitter Bootstrap

With the change to Twitter Bootstrap, the organization of pages has changed. Simple apps might continue to work okay, but any app that relied on the previous page structure to control styling or layout might need changes.

Dropped Django 1.3 support

We decided to drop Django 1.3 support to take advantage of the functionality offered in Django 1.4+, including bulk_create and override_settings. Additionally, with the release of Django 1.5, Django 1.3 is no longer supported by the Django developers.

Please read the Django 1.4 release notes for upgrade instructions, especially Backwards incompatible changes in 1.4.

Add Django 1.5 Support

RapidSMS 0.12.0 should work correctly with Django 1.5, and we encourage upgrading to Django 1.5 when possible.

Please read the Django 1.5 release notes for upgrade instructions, especially Backwards incompatible changes in 1.5.

We believe the most common changes affecting RapidSMS projects will be:

Removal of django.conf.urls.defaults

Make the following change to all of your `urls.py`:

```
-from django.conf.urls.defaults import *
+from django.conf.urls import patterns, url
```

Change to the url template tag

If any of your templates still use the old url template tag syntax, not quoting a literal url name, e.g.:

```
{% url url-name %}
```

for Django 1.5 they must be changed to quote the url name (or use a variable whose value is a url name):

```
{% url 'url-name' %}
```

If you wish to maintain compatibility with Django 1.4, you can add `{% load url from future %}` near the top of your template, e.g.:

```
{% load url from future %}

{% url 'url-name' %}
```

which will turn on support for quoted url names in Django 1.4, and be harmless in later Django releases.

Removed stale contrib apps

The following contrib applications have been removed:

- `rapidsms.contrib.ajax`: Old API used for communicating with the legacy router and no longer needed.
- `rapidsms.contrib.export`: Horribly insecure database export feature.
- `rapidsms.contrib.scheduler`: We officially adopted Celery for scheduling and asynchronous task processing. See Using Celery for Scheduling Tasks for more information.

If your project references these packages, you'll need to update your code appropriately.

New dependencies

Some of the contrib apps now use django-tables2 in place of the RapidSMS paginator utility to provide paging in tables. *django-tables2* requires less code to set up for common cases, and also allows eventually removing paginator from RapidSMS, so there's one less component to maintain.

The only app still using *djtables* is the locations app.

The messaging app uses django-selectable to provide autocompletion in an input field. *django-selectable* is a well-maintained, full-featured library for adding autocompletion in Django apps.

RAPIDSMS_TABS setting removed

The *RAPIDSMS_TABS* setting has been removed. This was used to configure the list of links displayed at the top of each page when using the RapidSMS templates. It was not very amenable to customization.

Starting in 0.12.0, the configurable links at the top of the page are generated by including a template, *rapidsms/_nav_bar.html*, which the RapidSMS project can override. Typically one would put list items there containing links. For example:

```
{% load url from future %}
<li><a href="{% url 'message_log' %}">Message Log</a></li>
<li><a href="{% url 'registration' %}">Registration</a></li>
<li><a href="{% url 'messaging' %}">Messaging</a></li>
<li><a href="{% url 'httptester' %}">Message Tester</a></li>
```

Region tags removed

These were in the base template.

Moved Message direction constants to model

The constant `rapidsms.contrib.messagelog.models.DIRECTION_CHOICES` has been moved to the `rapidsms.contrib.messagelog.models.Message` model. You may also refer to `Message.INCOMING` and `Message.OUTGOING` directly.

22.11.3 Test Coverage Report

With the addition of 26 tests, RapidSMS now has 136 automated unit tests with 82% (up from 72%) coverage.

22.12 RapidSMS 0.11.1 release notes

Welcome to RapidSMS 0.11.1! These release notes cover the new features 0.11.1.

22.12.1 What's New

- Fixed reference to `settings.CONTEXT_PROCESSORS` in the 0.10.0 migration guide.
- Added `send` stub to `rapidsms.backends.base.BaseBackend`.
- Fixed doc reference to Kannel backend path in *`INSTALLED_BACKENDS`*.

- Added documentation and tests for rapidsms.contrib.default, rapidsms.contrib.echo, and rapidsms.contrib.handlers.

22.13 RapidSMS 0.11.0 release notes

Welcome to RapidSMS 0.11.0! These release notes cover the new features 0.11.0.

22.13.1 What's New

- Use `load url from future` in templates to support Django 1.3-1.5. Thanks @miclovich!
- Moved the RapidSMS project template instructions to the main install page. See *Installing the RapidSMS project template*. Thanks @lemanal!
- Cleaned up a lot of core to be PEP8. Added instructions for using the `pep8` tool on the RapidSMS codebase. See *PEP 8 Style Guidelines Adherence*. Thanks @lemanal!
- TravisCI builds now run `coverage` and `pep8`. Here's an example from a recent build.
- **Simplified testing with introduction of `RapidTest` class. See *RapidTest*. This includes:**
  - Add `RapidTest` and `RapidTransactionTest` base classes
  - Modify `get_router()` to return an instantiated object rather than a class
  - Update `TestRouterMixin` to patch `RAPIDSMS_ROUTER` directly with instantiated `TestRouter`
  - Remove global variables/state from `TestRouter`
  - Clean up documentation to focus more on `RapidTest` and `RapidTransactionTest` classes
  - Remove `MockBackendRouter` test class
  - Update `TestScript` to use `RapidTest`
- Moved official RapidSMS version to `rapidsms.__version__`. Reference it directly from docs and setup.py.
- Added coverage reports to core test suite. See *RapidSMS core test suite*.
- Moved `rapidsms` module to root level of repository.
- Added `docs` environment to tox setup.
- Documented django-celery 3.0+ dependency.
- Removed `rapidsms.skeleton` package and `startproject` command override See *Installing the RapidSMS project template* to use the new project template.
- Removed `runrouter` management command.
- Fixed a few typos in the Messaging API docs.

22.14 RapidSMS 0.10.0 release notes

Welcome to RapidSMS 0.10.0!

These release notes cover the new features in 0.10.0, as well as some *backwards-incompatible-changes* you'll want to be aware of when upgrading from RapidSMS 0.9.6a or older versions. We also provide a migration guide to help you port your 0.9.6 projects and apps to 0.10.0 to take advantage of the new features.

22.14.1 Overview

RapidSMS 0.10.0's focus has mostly been on decoupling the RapidSMS route process in several key places to begin processing all SMSes in normal HTTP requests. This also includes making it possible to swap the Router class that RapidSMS uses via a setting in the settings file. The key changes are as follows:

- Improved documentation (what you're reading now!)
- Improved test coverage and made it easier to test your RapidSMS apps.
- Added support for `django.contrib.staticfiles`.
- Removal of the bucket, email, irc, gsm, and smtp backends.
- Dividing the Router logic into `BaseRouter` and `BlockingRouter` classes, and the addition of a Celery-powered router, `CeleryRouter`.
- Removal of the legacy persistent threaded router.

22.14.2 What's new in RapidSMS 0.10.0

The major highlights of RapidSMS 0.10.0 are:

A new router

RapidSMS 0.10.0 supplies one built-in router, **BlockingRouter**. This is the default router that processes messages in real time.

We also support creation of custom router classes. All routers should extend from the `BaseRouter` class.

Removal of threaded router

In 0.9.x, the RapidSMS router used Python's threading module to encapsulate backends into independent threads. Using this model, backends can operate independently from one another, blocking for I/O and waiting for external service calls. Many of the original backends operated in this way. For example, `rapidsms.backends.http` started a HTTP server to listen on a specified port and `rapidsms.backends.gsm` communicated directly with a GSM modem. While this method provided RapidSMS with a routing architecture, the need for a non-threaded system grew due to the following reasons:

- Thread interaction was complicated and not always intuitive.
- If the route process died unexpectedly, all backends (and hence message processing) were brought offline.
- Automated testing was difficult and inefficient, because the router (and all its threads) needed to be started/stopped for each test.

Added RAPIDSMS_ROUTER setting

RapidSMS now allows you to specify the primary router class to use by defining *`RAPIDSMS_ROUTER`* in settings. This defaults to `rapidsms.router.blocking.BlockingRouter`, but you can change this in settings.py:

```
RAPIDSMS_ROUTER = 'myproject.router.MyRouter'
```

Added `get_router()` utility

A new utility function, `get_router`, provides the ability to retrieve the settings-defined router. This helper function allows your app to remain router independent:

```
from rapidsms.router import get_router

def send(recipient, text):
    router = get_router()()
    router.handle_outgoing(text, recipient.default_connection)
```

Backends are Django apps

RapidSMS backends are now apps (rather than modules) in the `rapidsms.backends` directory. RapidSMS provides two built-in backend apps: `http` and `kannel`. We have *completely removed all other backends* from the RapidSMS core.

We also support creation of custom backend apps. Backend classes should extend from the classes found in `rapidsms.backends.base`.

Added `MockBackendRouter` class

`MockBackendRouter` is a unit test mix-in class that provides a mock backend to use with the `BlockingRouter`. The following example from `contrib.messaging` illustrates how you can test that inbound messages route to the mock backend outbox.

```
from django.test import TestCase
from rapidsms.tests.harness.base import MockBackendRouter

class MessagingTest(MockBackendRouter, TestCase):

    def setUp(self):
        self.contact = self.create_contact()
        self.backend = self.create_backend({'name': 'mock'})
        self.connection = self.create_connection({'backend': self.backend,
                                                  'contact': self.contact})

    def test_ajax_send_view(self):
        """
        Test AJAX send view with valid data
        """
        data = {'text': 'hello!', 'recipients': [self.contact.id]}
        response = self.client.post(reverse('send_message'), data)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(self.outbox[0].text, data['text'])
```

Updated `TestScript`

Prior to 0.10.0, `TestScript` would instantiate the route process (with blocking backends) to allow for testing of the entire routing stack. This was a useful function, but in practice was unstable and caused tests to hang indefinitely. In 0.10.0, `TestScript` has been updated to work with `BlockingRouter`, and it functions much in the same way as before. Here's an example testing the EchoApp:

```
class EchoTest(TestScript):
    apps = (EchoApp,)

    def testRunScript(self):
        self.runScript("""
            2345678901 > echo?
            2345678901 < 2345678901: echo?
        """)
```

22.14.3 Backwards-incompatible changes in RapidSMS 0.10.0

In the goal of improving the RapidSMS core, we have made a number of backwards-incompatible changes. If you have apps written against RapidSMS 0.9.6 that you need to port, see our migration guide.

Supporting Django 1.3+

RapidSMS is no longer compatible with any version of Django prior to 1.3.

Static media handled by django.contrib.staticfiles

RapidSMS 0.10.0 supports out-of-the-box use of `django.core.staticfiles` (included by default in Django 1.3.x and above). The `rapidsms.urls.static_media` module has been removed in favor of using this app. New projects generated using `rapidsms-admin.py startproject` are automatically configured to work with staticfiles. See the migration guide for more information on upgrading existing projects.

Removal of backends

We removed several rarely-used or outdated backend packages from the core:

- `rapidsms.backends.bucket`
- `rapidsms.backends.email`
- `rapidsms.backends.irc`
- `rapidsms.backends.gsm`
- `rapidsms.backends.smtp`

Removal of `rapidsms.contrib.ajax` app

The `rapidsms.contrib.ajax` app has been removed.

Removal of send_message

Prior to 0.10.0, rapidsms.contrib.messaging contained a utility function to send a message to the Router process. This relied on the contrib.ajax's call_router function to pass messages to the Router via the ajax app running in the Router thread. send_message has been removed and you should now use rapidsms.router.send (see *Sending Messages*). Using send_message will now raise an exception:

```
>>> from rapidsms.contrib.messaging.utils import send_message
>>> send_message(conn, "hello?")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "../rapidsms/lib/rapidsms/contrib/messaging/utils.py", line 2, in send_message
    raise DeprecationWarning("rapidsms.contrib.messaging.utils is deprecated")
DeprecationWarning: rapidsms.contrib.messaging.utils is deprecated
```

Scheduler refactor

rapidsms.contrib.scheduler still exists, but is currently incompatible with 0.10.0. We plan to support the scheduler in the next minor RapidSMS release.

22.15 Migrating your Project from RapidSMS 0.9.6 to 0.10.0

RapidSMS 0.10.0 breaks compatibility with 0.9.6 in some areas. This guide will help you port 0.9.6 projects and apps to 0.10.0. We describe the changes that most users will need to make, as well as *less-common compatibility issues* to look for if your code is still breaking.

Note: See the 0.10.0 release notes. That document explains the new features in RapidSMS more deeply; this porting guide is more concerned with helping you quickly update your code.

22.15.1 Upgrade to Django>=1.3

RapidSMS 0.10.0 will only support Django 1.3 or newer.

- Django 1.3 release notes
- Django 1.4 release notes

22.15.2 Choose a router

The global router located at rapidsms.router.Router has been removed. All routers now extend from a new base class, rapidsms.router.base.BaseRouter. RapidSMS 0.10.0 offers one built-in router option, as well as support for third-party routers.

A new setting, *RAPIDSMS_ROUTER*, allows you to specify the primary router class to use. A new utility function, rapidsms.router.get_router, retrieves the router defined in *RAPIDSMS_ROUTER* so that your project can remain router-independent.

BlockingRouter

`rapidsms.router.blocking.BlockingRouter` is the default value for *RAPIDSMS_ROUTER*. This router processes messages within the HTTP request/response cycle and does not perform any additional queuing of messages. It simplifies testing and development, and also works well will Kannel or other message gateways for small- to medium-sized RapidSMS sites. You may wish to specify the router in your settings file explicitly as follows:

```
RAPIDSMS_ROUTER = 'rapidsms.router.blocking.BlockingRouter'
```

Custom Router

`BlockingRouter` is sufficient for many applications, but applications handling hundreds of messages per second or more may require custom routing logic. If you are running such an application, you might consider writing your own router or finding a third-party router that fits your needs. All custom routers should extend from `rapidsms.router.base.BaseRouter`.

22.15.3 Update old backends

RapidSMS backends are now apps (rather than modules) in the `rapidsms.backends` directory. You may not need to update your imports the main backend classes live in the `__init__.py` file of their app.

RapidSMS provides two built-in backend apps: `http` and `kannel`. We have completely removed these backends from the RapidSMS core:

- `rapidsms.backends.bucket`
- `rapidsms.backends.email`
- `rapidsms.backends.gsm`
- `rapidsms.backends.irc`
- `rapidsms.backends.smtp`

Upgrading to Kannel

If you used PyGSM or one of the other non-HTTP backends, you must update your code to use a different backend. For many use cases, the `kannel` backend will be a good replacement for the `gsm` backend. For more information about configuring RapidSMS to use Kannel, please see the Kannel backend documentation.

Other backends

If none of the new backends suit the needs of your project, you may write a custom backend app. Backend classes should extend from the classes found in `rapidsms.backends.base`.

22.15.4 Remove `rapidsms.contrib.ajax`

We have removed the `rapidsms.contrib.ajax` app. You should update your settings, URL configuration, and other project code to reflect this change.

1. Remove `rapid.contrib.ajax` from `settings.INSTALLED_APPS`.
2. Remove `rapid.contrib.ajax` from `settings.TEST_EXCLUDED_APPS`.

3. Remove ajax URLs from your URL configuration file.

22.15.5 Remove calls to `rapidsms.contrib.messaging.send_message`

As the method `rapidsms.contrib.messaging.send_message` relied upon `rapidsms.contrib.ajax` functionality, it has been deprecated and will raise a warning when used. All calls to `send_message` should be replaced with calls to `rapidsms.router.send` (see *Sending Messages*).

22.15.6 Use `django.contrib.staticfiles`

The `rapidsms.urls.static_media` module has been removed from RapidSMS 0.10.0 in favor of using `django.core.staticfiles` (which is included by default in Django 1.3.x and above). To upgrade your project, take the following steps:

1. Add `"django.contrib.staticfiles"` to `settings.INSTALLED_APPS`.
2. Add `"django.core.context_processors.static"` to `settings.TEMPLATE_CONTEXT_PROCESSORS`.
3. Remove references to `rapidsms.urls.static_media` from your URL configuration or other places in your project. The location of static files should now be handled by the classes listed in `settings.STATICFILES_FINDERS`. By default, RapidSMS requires these finders:

```
STATICFILES_FINDERS = (
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
)
```

4. Define the URL prefix for static files in `settings.STATIC_URL`. This value should be distinct from `settings.MEDIA_URL`.
5. Define the absolute path of the directory to which static files should be collected in `settings.STATIC_ROOT`.
6. Each app should maintain its static media in the `static/` subfolder. (If you have any other directories in which static files are maintained, list them in `settings.STATICFILES_DIRS`.) We have adopted the convention of keeping the app's static files in a subfolder of `static/` with the same name as the app. For example, the static file located at `myapp/static/myapp/stylesheets/myapp.css` will be available at `{{ STATIC_URL }}myapp/stylesheets/myapp.css`.
7. Ensure that URLs to static files in your templates use `{{ STATIC_URL }}` rather than `{{ MEDIA_URL }}` to locate static files.

22.15.7 Refactor stateful applications to use the database or cache

Some RapidSMS applications in the community may use the `App` instance (or module-level variables in Python) to store persistent state information about the application. Now that routers are constructed and destroyed for every incoming message, state information stored on apps will not persist between requests. Code that makes this assumption should be refactored to use a database or cache to store data that need to persist between requests.

22.15.8 Scheduler refactor

`rapidsms.contrib.scheduler` is currently incompatible with v0.10.0. Until we release a compatible version, we recommend investigating cron-style methods or using Celery's periodic tasks.

22.15.9 Less-common changes

The following changes are smaller, more localized changes. They should only affect more advanced users, but it's probably worth reading through the list and checking your code for these things.

TestScript

Prior to 0.10.0, `TestScript` would instantiate the routing process (with blocking backends) to allow for testing of the entire routing stack. In the new release, `TestScript` has been updated to work with `BlockingRouter`. In most cases, the changes to the `TestScript` class should not affect how you write your test code.

22.16 RapidSMS 0.9.6 release notes

Welcome to RapidSMS 0.9.6!

22.16.1 Extensible Models

The ExtensibleModelBase class lets you define models that link to other models, creating a chain of inheritance. By defining django models in:

```
<extending_app>/extensions/<app_to_extend>/<model_to_extend>.py
```

you can get new top-level properties on the <model_to_extend> object.

In order for your base model class to support this you must add the following line:

```
__metaclass__ = ExtensibleModelBase
```

Additionally the django models defined in the file should be declared abstract, since they won't be instantiated.

See the *Contact* model in the rapidsms core for an example of a model that can be extended, and the contact.py file in the locations/extensions/rapidsms contrib app folder for an example of it being extended. The end result of these two classes is that a .location is available on instances of Contact models as a foreign key to the Location table.

22.16.2 Django App Settings

A handy little module that lets you define default settings within your django app and import them from a second settings.py. There is a good README in the django-app-settings submodule, but from a practical use standpoint, all you have to remember is to include a settings.py in your root app/ directory with any app-specific settings and import your settings from rapidsms.conf instead of django.conf and everything should just work. It's a good practice to use app-specific prefixes for your settings to avoid conflicts.

22.16.3 The Webapp

The *apps/webapp* is gone, and *lib/rapidsms* _is_ the new webapp. If you are new to rapidsms, just remember that lib/rapidsms is the app that is the main entry point to the webui, and that's where the base urls and templates are stored.

22.16.4 Different Classes

(these are also described in the porting apps section below)

Reporter app's commonly used Reporter, PersistantBackend, and PersistantConnection models are gone!

They have been replaced by rapidsms.model's Contact, Backend, and Connection models (which are all persistent, so app authors no longer need to worry about keeping track of such things).

lib/rapidsms/message.py's Message is gone! Instead, there are MessageBase, IncomingMessage, and OutgoingMessage classes in lib/rapidsms/messages/

rapidsms.app.App has been replaced by rapidsms.apps.base.AppBase

22.16.5 Porting apps to work with the latest RapidSMS code

Views

RapidSMS's wrapper of Django's render_to_response method is gone, so change the import line from:

```
from rapidsms.utils import render_to_response
```

to:

```
from django.template import RequestContext
from django.shortcuts import render_to_response
```

and change the render_to_response method calls from:

```
return render_to_response(req, "myapp/mytemplate.html")
```

to:

```
return render_to_response("myapp/mytemplate.html", context_instance=RequestContext(req))
```

Apps

rapidsms.app.App —> rapidsms.apps.base.AppBase

from reporters.models import Reporter, PersistantConnection —> from rapidsms.models import Contact, Connection

from rapidsms.message import Message —> from rapidsms.messages import IncomingMessage, OutgoingMessage

Handlers

from rapidsms.contrib.handlers import KeywordHandler —> from rapidsms.contrib.handlers.handlers.keyword import KeywordHandler

22.17 RapidSMS Roadmap

Below you'll find a rough outline of planned milestones and releases for RapidSMS. For a list of official releases, please see the Release Notes.

22.17.1 v0.16.0

- *Release date:* November 6, 2013
- Misc. fixes

22.17.2 New RapidSMS website

- *Expected release date:* July, 2013

Goals

- Revamp RapidSMS website with new design
- Highlight high level stories of current installations with pictures and maps
- Provide a page to track 3rd party reusable apps and backends
- Blog syndication (community page)
- Migrate existing content to new platform

22.17.3 v0.15.0

- **Tutorial and contributing docs**
- *Release date:* June 1, 2013
- v0.15.0 GitHub Milestone

Goals

- Write a tutorial similar to the Django tutorial for beginners
- Finish development on outstanding core features and bugs

22.17.4 v0.14.0

- **Scheduling and deployment**
- *Release date:* April 30, 2013
- v0.14.0 GitHub Milestone

Goals

- Review and analyze cloud hosting providers
- Write comprehensive deployment documentation for chosen providers
- Provide instructions and scripts to deploy project in a few simple steps

22.17.5 v0.13.0

- **Bulk messaging**
- *Release date:* April 10, 2013
- v0.13.0 GitHub Milestone

Goals

- Update scheduling architecture based on community proposal
- Load testing of message handling and routing functionality
- Identify bottlenecks and create plans for improving performance
- Write documentation for users intending to operate RapidSMS at scale
- Integrate 3rd party service providers like Vumi
- Implement the Bulk Messaging API
- Finalize and merge the Vumi backend pull request

22.17.6 v0.12.0

- **Bootstrap and contrib update**
- *Release date:* March 21, 2013
- v0.12.0 GitHub Milestone

Goals

- Use Twitter Bootstrap in core, based on the community proposal
- Update contrib applications to meet base standard as per this review

22.17.7 v0.11.0

- **PEP8 and testing**
- *Release date:* December 31, 2012
- RapidSMS 0.11.0 release notes
- v0.11.0 GitHub Milestone

Goals

- Update and simplify test harness
- Add coverage/PEP8 usage guide and documentation

22.17.8 v0.10.0

- **New routing**
- *Date released:* November 23, 2012
- RapidSMS 0.10.0 release notes
- v0.10.0 GitHub Milestone

Goals

- Introduce new-routing architecture
- Improve documentation

22.17.9 v0.9.6a

- *Date released:* October 19, 2010
- RapidSMS 0.9.6 release notes

CHAPTER 23

RapidSMS internals

Documentation for people hacking on RapidSMS itself.

23.1 RapidSMS 1.0 Roadmap

This document describes the high level goals and schedule for releasing RapidSMS 1.0. It was originally created by Colin Copeland and Tobias McNulty in collaboration with UNICEF Innovation. However, the document is open to the greater RapidSMS community for discussion, comments, and other feedback.

23.1.1 Design Philosophies

- **Encourage community involvement.** New and long term RapidSMS users should feel welcomed in the community. Introductory materials, such as tutorials and how-to documentation, should be written to help beginners. Standards and design patters should be in place to make the development environment more consistent.
- **Be more Django-like.** RapidSMS is, for the most part, a set of Django applications with a message processing component. RapidSMS should be packaged like every other Django app and follow the communities best practices.
- **Improve test coverage.** Every good Python package should provide a robust test suite for coverage and regression testing. New core code should always include tests before being merged into master.
- **Write better documentation.** The RapidSMS project should provide consistent and readable documentation. The documentation should be written in a maintainable format (ReST) and we should aim to improve it as often as possible.
- **Batteries included.** The bar needs to be raised for the contrib applications. RapidSMS should provide you with enough tools out of the box to hit the ground running.
- **Guidelines for maintenance and scaling.** Deployment examples and best practices should be provided to ease the transition to hosted environments.

23.1.2 Roadmap

Month	Focus
July	Develop roadmap and plan
August	Improve test suite against core
September	Improve and cleanup documentation
October	Encourage ongoing developer participation in RapidSMS itself
November	Clean up core and prepare for v1.0 release
December	Provide community blessed way to package and distribute pluggable RapidSMS apps
January	Optimize core for scaling
February	Revamp RapidSMS website
March	Build extensible implementations of key RapidSMS core / contrib apps
April	Release 1.0
May	Create and document RapidSMS deployment methods

Develop roadmap and plan

- Conduct assessment of RapidSMS code and community
- Begin developing community document outlining strategy and workplan for 1.0 release
- Survey 3rd-party integration points and discuss plan for core modification
- Delivery: End of month 1

Improve test suite against core

- Setup and maintain Jenkins CI for RapidSMS
- Set standard for test coverage
- Set the precedent for including unit tests both in projects and RapidSMS itself
- Delivery: End of month 2

Improve and cleanup documentation

- Write documentation for existing core functionality
- Installation instructions
- Configuration and deployment instructions
- Take down or redirect links to old documentation
- Delivery: End of month 3

Encourage ongoing developer participation in RapidSMS itself

- Define a structured way to contribute to RapidSMS and how to help out
- Designate roles (“Release manager”) & encourage individuals to champion features
- Organize RapidSMS development sprints
- Delivery: End of month 4

Clean up core and prepare for v1.0 release

- Push development of new router
- Cleanup and document built-in backends
- Determine release schedule that focuses on releasing early and often
- Set release date for 1.0 and create publicity for the release
- **UNICEF Deliverables:**
  - Provide list of existing RapidSMS projects and apps
- Delivery: End of month 5

Provide community blessed way to package and distribute pluggable RapidSMS apps

- Identify existing apps and projects that would be good candidates for packaging
- Survey the community on the list for such apps
- Provide documentation for packaging apps and distributing to community
- Provide guidelines for making apps extensible
- Build small apps as examples using proposed packaging guidelines
- Provide support packaging using the provided examples, test coverage and documentation
- Identify overlap of different projects, e.g., two apps that do group management
- **UNICEF Deliverables:**
  - Sign off on new website design and functionality
  - Server for RapidSMS website
- Delivery: End of month 6

Optimize core for scaling

- Load testing of message handling and routing functionality
- Identify bottlenecks and create plans for improving performance
- Write documentation for users intending to operate RapidSMS at scale
- Work with Evan to integrate 3rd party service providers like Vumi
- Delivery: End of month 7

Revamp RapidSMS website

- Highlight high level stories of current installations with pictures and maps
- Provide a page to track 3rd party reusable apps and backends
- Blog syndication (community page)
- Migrate existing content to new platform
- Begin marketing new release

- **UNICEF Deliverables:**
  - Information gathering and content writing for featured case studies on website
- Delivery: End of month 8

Build extensible implementations of key RapidSMS core / contrib apps

- locations
- groups
- Delivery: End of month 9

Release 1.0

- Write a tutorial similar to the Django tutorial for beginners
- Finish documentation for new core features
- Write release notes for v1.0
- Finish development on outstanding core features and bugs
- Delivery: End of month 10

Create and document RapidSMS deployment methods

- Review and analyze cloud hosting providers
- Write comprehensive deployment documentation for chosen providers
- Provide instructions and scripts to deploy project in a few simple steps
- Delivery: End of month 11

23.2 mHealth Interoperability Survey

Stemming from our discussions at the Rwanda mHealth System Design Workshop, this brief survey outlines the possible integration points between RapidSMS and various mHealth open source projects.

23.2.1 OpenMRS

Open Medical Record System (OpenMRS) is a software platform and a reference application which enables design of a customized medical records system with no programming knowledge.

- OpenMRS website, OpenMRS Developers Portal
- Date created: 2004
- Architecture: Java/MySQL
- Features: Patient database with detailed visit history and metrics.
- **RapidSMS Interoperability:**
  - API: Internal Java API?

  - Patient backend? via MySQL?

23.2.2 Vumi/praekelt

A super-scalable conversation engine for the delivery of SMS.

- Vumi website, Vumi Developers Portal
- Architecture: Python
- Features: Message sending framework. Can write Vumi-level applications for deeper integration. Can be used as a hosted service?
- RapidSMS Interoperability: support has been integrated into RapidSMS 0.13

23.2.3 DHIS

The District Health Information System (DHIS) 2 is a tool for collection, validation, analysis, and presentation of aggregate statistical data, tailored (but not limited) to integrated health information management activities.

- DHIS website, DHIS Developers Portal
- Date created: 2008
- Architecture: Java frameworks, such as the Spring Framework, Hibernate, Struts2, Maven, and JUnit.
- Features: Data analysis and aggregation tool. Mapping features.
- **RapidSMS Interoperability**
  - API: "Rich Web API" - REST?
  - Probably easiest to push data to DHIS2.
  - Idea: Django app to model DHIS2 data structures and push on demand.

23.2.4 MOTECH

The MOTECH Suite is a set of Open Source technology components from a consortium of partners who have recognized that their complementary software efforts can address the core needs of mHealth.

- MOTECH website
- Date created: 2008
- Architecture: MOTECH is a modular, event driven system written in Java.
- Features: A framework with built in support for SMS registration, IVR, scheduled messages, reports.
- **RapidSMS Interoperability**
  - REST API?

23.3 How to Make RapidSMS Tutorial Videos

This "how to" is for Linux users, primarily Ubuntu users. It will work in 10.4, Karmic Koala. The goal is to be able to record your screen and voice, transcode it into a format that youtube understands and then upload your video to youtube.

23.3.1 Recording Sound and Video

First you will need a piece of software to record your screen and voice. There are many of these but not all are very well supported. recordMyDesktop works well and is simple and configurable. To get this you can go to the Applications Menu>Ubuntu Software Center and type "gtk rec" gtk-recordMyDesktop will come up first and you can double click on it to install. It will then be in your Applications Menu under "Sound & Video."

Alternatively just type:

```
sudo apt-get install gtk-recordMyDesktop
```

There is a tutorial on how to install, use and configure recordMyDesktop here:

http://www.youtube.com/watch?v=HaAXW67SUgk

23.3.2 Turning your .ogv into .avi

Next you will want to take the .ogv format that recordMyDesktop creates and turn it into a format that youtube will accept. There are many pieces of software that can do this and a good one to turn it into .avi is mencoder.

To install mencoder type:

```
sudo apt-get install mencoder
```

Once it is installed type (changing foo to the path and name of your video):

```
mencoder foo.ogv -o foo.avi -oac mp3lame -lameopts fast:preset=standard -ovc lavc -lavcopts vcodec=mp
```

There is a tutorial on how to convert using mencoder and then upload to youtube here:

http://www.youtube.com/watch?v=VuhYV0voL3M

23.3.3 Posting to youtube

Create a youtube account (or use your existing one) and follow the tutorial above or any of the billion other tutorials on the internet to upload your video. Tag your video with both "rapidSMS" and "tutorial." Then share the video with the RapidSMS Developer user on youtube and post a message to the mailing list about your new video. It will get favorited by the community and show up on the channel.

The youtube channel is here: http://www.youtube.com/user/rapidsmsdev

For access to the RapidSMSdev youtube account ask Merrick.

CHAPTER 24

RapidSMS License

The RapidSMS code is licensed under BSD (see LICENSE). UNICEF acts as the holder of contributor's agreements. If you wish to contribute to the RapidSMS codebase as an individual or an organization, you will need to sign the contributor's agreement.

24.1 Contributor Licence Agreements (CLAs)

The license agreement is a legal document in which you state you are entitled to contribute the code/documentation/translation to RapidSMS and are willing to have it used in distributions and derivative works. This means that should there be any kind of legal issue in the future as to the origins and ownership of any particular piece of code, UNICEF has the necessary forms on file from the contributor(s) saying they were permitted to make this contribution.

This is a pure license agreement, not a copyright assignment. You still maintain the full copyright for your contributions. You are only providing a license to UNICEF to distribute your code without further restrictions. This is not the case for all CLA's, but it is the case for the one we are using.

- Sign Individual CLA | List of individual signers
- Sign Corporate CLA | List of corporate signers

24.2 History

The commit history and discussion around the license and agreements can be found below:

- rapidsms Google Group discussion
- Git commit with the change

Release: v0.21.0. (Installation, Release Notes)

Getting Started

- Overview
- Installation
- Tutorial
- Getting help
- Release notes and upgrading instructions

Architecture

- RapidSMS architecture overview
- **Router:** Overview | Messaging API | BlockingRouter | CeleryRouter | DatabaseRouter
- **Applications:** Overview | Community apps
- **Backends:** Overview | Kannel | Vumi | Database | Custom

The development process

- Virtual environments
- Settings
- Internationalization
- Extending core RapidSMS models
- Front end - Creating a web interface for your app
- Logging
- Testing
- Scheduling Tasks with Celery
- Packaging a RapidSMS application for re-use

Provisioning & Deploying Your Project

- Overview
- Planning
- Provisioning
- Deploying
- Scaling

The RapidSMS open-source project

- *Developing RapidSMS*
- *The RapidSMS Community*
- *License* - How RapidSMS is licensed

RapidSMS contrib applications

- default - Sends a pre-defined default response to messages that are not handled by any other application.
- echo - A collection of two simple handlers that can assist you in remote debugging.
- handlers - Extensible classes that help you create RapidSMS applications quickly.
- httptester - Helps you test your project by sending fake messages to RapidSMS to see how it responds.
- locations - Defines the *Location* model, which helps you map custom locations and points in your project.
- messagelog - Maintains a record of all messages sent and received by RapidSMS.
- messaging - Provides a web interface through which you can send messages to Contacts.
- registration - Provides a web interface for creating, updating, and deleting RapidSMS contacts.

CHAPTER 25

Indices and tables

- genindex
- modindex
- search

r

G

H

I

K

L

M

N

O

P

R

S

T