rapidsms-healthcare Documentation

Release 0.1.0

Caktus Consulting Group

March 26, 2013

CONTENTS

rapidsms-healthcare is a reusable Django application for managing healthcare provider and patient records for building RapidSMS applications. The goal is to create a common API for storing and accessing these records and have configurable storage backends for the data itself. For instance on site might store data using a SQL database via the Django ORM while another might store the data in OpenMRS. Additional Django/RapidSMS applications can use this API to store and retrive data without knowning how it will be stored.

CHAPTER

ONE

DEPENDENCIES

rapidsms-appointments currently runs on Python 2.6 and 2.7 and requires the following Python packages:

- Django >= 1.4
- RapidSMS >= 0.11.0

CHAPTER

TWO

DOCUMENTATION

Documentation on using rapidsms-healthcare is available on Read The Docs.

CHAPTER

THREE

RUNNING THE TESTS

With all of the dependancies installed, you can quickly run the tests with via:

```
python setup.py test
```

or:

```
python runtests.py
```

To test rapidsms-healthcare in multiple supported environments you can make use of the tox configuration.:

```
# You must have tox installed
pip install tox
# Build default set of environments
tox
# Build a single environment
tox -e py26-1.4.X
```

CHAPTER

FOUR

LICENSE

rapidsms-healthcare is released under the BSD License. See the LICENSE file for more details.

CHAPTER

FIVE

CONTRIBUTING

If you think you've found a bug or are interested in contributing to this project check out rapidsms-healthcare on Github. A full contributing guide can be found in the online documentation.

Development sponsored by Caktus Consulting Group, LLC.

CHAPTER

SIX

CONTENTS

6.1 Project Overview

As the name implies rapidsms-healthcare aims to solve a problem with storing healthcare related data for RapidSMS deployments. Currently there is no canonical data model for storing patient or healthcare worker data in RapidSMS. This means that each deployment tends to create its own model for this data. Applications that are then built to use this data are tied to the model which hurts the reusability of these applications.

At the same time with development of external electronic medicial record systems such as OpenMRS, future deployments may choose to use these solutions rather than storing data via the Django ORM. So rather than try to create a canonical representation of the data, this project intends to create a seperation between the storage and access of the data. Applications which need to access or store patient data will do so through a common API without knowledge of the storage allowing the same code to be used in a deployment which uses the ORM and another which uses OpenMRS.

6.1.1 Goals

The development of rapidsms-healthcare is still in the very early stages but the primary goals for both the current and future work are

- Client API for storing and retriving patient and provider records
- Default storage backend using the Django ORM
- Additional storage backend using OpenMRS
- Documentation guide/standards for creating additional storage backends
- Suite of applications known to work with this patient API

6.1.2 Related Projects

There two current projects which attempt to provider flexible data models for patient records

- https://github.com/unicefuganda/rapidsms-healthmodels
- https://github.com/ewheeler/rapidsms-people-app

While the flexibility is helpful for a singular deployment, it contributes to the difficulty when trying to create applications which can be used on a broad set of deployments.

6.2 Getting Started

Here you will find the necessary steps to install and initial configure the rapidsms-healthcare application.

6.2.1 Requirements

rapidsms-healthcare requires Python 2.6 or 2.7. Python 3 is not currently supported but is planned in the future as Django and RapidSMS support for Python 3 increases. It also requires the following packages:

- Django >= 1.4
- RapidSMS >= 0.11.0

6.2.2 Installation

Stable releases of rapidsms-healthcare can be found on PyPi and pip is the recommended method for installing the package:

```
pip install rapidsms-healthcare
```

6.2.3 Configuration

The storage and retrieval of healthcare related data is configured by the *HEALTHCARE_STORAGE_BACKEND* setting. If you are using the default storage backend you need to change your `INSTALLED_APPS` to include:

```
INSTALLED_APPS = (
    # Other apps go here
    'healthcare.backends.djhealth',
)
```

If you are using a different backend then you can skip this step. The Django backend uses South to manage possible future changes to the schema. While not required if you are using South in your project then you can create the tables needed for the backend via:

```
python manage.py migrate djhealth
```

If you are not using South then you can create the tables via:

```
python manage.py syncdb
```

Note: While using South is optional, it is highly recommended. If you are not using South then you may need to apply future schema change yourself. When needed these will be noted in the release notes.

6.2.4 Next Steps

- For information on storing and retrieving data you can go to the *usage documentation*.
- For information on creating a custom storage backend please see the *backend API documentation*.
- If you have found and bug or would like to contribute a feature you can go to the *contributing guide*.

6.3 Basic Usage

The primary usage of this application is for other RapidSMS applications to have a common way to store and retrive patient/provider data even though each project which deploy the application may use different methods of actually storing the data.

Applications built with rapidsms-healtcare should use the functionality provided by `healthcare.api`. The `client` object in the module is primary entry point for accessing the healthcare data. It provides basic CRUD (Create/Update/Delete) operations for the patient and provider data:

```
from healthcare.api import client

patient = client.patients.create(name='Joe', sex='M')
```

6.3.1 Available Backends

Where this data is stored configured by the *HEALTHCARE_STORAGE_BACKEND* setting. Below are all of the available backends included in the rapidsms-healthcare distribution.

DjangoStorage

Path: `'healthcare.backends.djhealth.DjangoStorage'`

This is the default storage backend. It stores the patient and provider information using the Django ORM. To use this backend you must include `'healthcare.backends.djhealth'` in your `INSTALLED_APPS` setting to create the necessary tables.

Warning: Though the default storage backend uses the Django ORM, developers should resist all temptation to access the models directly (including creating FKs in additional models) as that will break the portability of the application.

DummyStorage

Path: `'healthcare.backends.dummy.DummyStorage'`

This backend stores the data in local memory. This backend is indended only for testing and should never be used in a production setting.

6.3.2 Patient Information

Operations on patient data are done through `client.patients`. Currently the backend supports *create*, *get*, *update*, *filter*, *delete*, *link*, and *unlink*:

```
from healthcare.api import client

# Create new patient
patient = client.patients.create(name='Joe', sex='M')

# Update the patient's name
client.patients.update(patient['id'], name='Jack')

# Refetch the patient record
```

```
patient = client.patients.get(patient['id'])

# Get patients with name containing 'Ja'
patients = client.patients.filter(name__like='Ja')

# Delete a patient record
client.patients.delete(patient['id'])
```

Patient Data Model

Patients currently store the following pieces of data [1]:

Name	Type	Description
id	String	Globally unique identifier
name	String	A readable name/identifier
sex	String (M/F)	Patient's sex
birth_date	Date	Date the patient was born
death_date	Date	Date the patient died
location	String	Identifier for the patient's location
created_date	Datetime	Time when the record was created
updated_date	Datetime	Time when the record was last updated
status	String (A/I)	Flag to denote if the record is currently active

The `location` field might store the name of the location or an identifier for another location/facility registry.

`patients.create`

`patients.create` adds a new patient record to the data store. The arguments for this function are passed to the backend to store on the record. The patient data is returned as a dictionary and contains additional fields which are generated by the backend: `id`, `created_date` and `updated_date`.

`patients.update`

`patients.update` takes the id of the patient along with arguments to be passed to the backend to update. This returns a boolean to note whether a matching patient was found and updated.

`patients.get`

`patients.get` returns a patient's data as dictionary for the given id. If no matching patient was found this will raise a `PatientNotFound` exception.

Patients can also be associated with external ids using the *link* method. You can retrieve these users using `get` by passing the source name of the identifier.:

```
from healthcare.api import client

# Create new patient
patient = client.patients.create(name='Joe', sex='M')

# Associate patient with an external ID
client.patients.link(patient['id'], '123456789', 'NationalID')
```

[1] Required fields are bold and generated values are in italics.

```
# Refetch the patient record using national id
patient = client.patients.get('123456789', source='NationalID')
```

`patients.filter`

`patients.filter` returns a list of matched patient data dictionaries. If there are no matches then it will be an empty list. Additional details on filtering expressions is given below.

`patients.delete`

`patients.delete` takes the id of the patient and returns a boolean to note whether a matching patient was found and deleted.

`patients.link` and `patients.unlink`

The `patient.id` is generated by the backend and cannot be controlled by the application. `patients.link` and `patients.unlink` are used to manage associations between patients and additional identifiers used by the application. These might be identifiers created internally by application, assigned by health care facilities or national identifiers. To create a new association you need patient id, the additional id, and a name for the source of the id. The additional identifiers should be unique for their source.:

```
from healthcare.api import client

# Create new patient
patient = client.patients.create(name='Joe', sex='M')

# Associate patient with an external ID
client.patients.link(patient['id'], '123456789', 'NationalID')
```

`patients.unlink` is used to remove this association.:

```
# Continued from the above example...

# Remove patient's external ID
client.patients.unlink(patient['id'], '123456789', 'NationalID')
```

The `patients.link` and `patients.unlink` both return booleans to denote whether the association creation/deletion was successful.

6.3.3 Provider Information

Operations on patient data are done through `client.providers`. Currently the backend supports `create`, `get`, `update`, `filter` and `delete`:

```
from healthcare.api import client

# Create new provider
provider = client.providers.create(name='Joe')

# Update the providers's name
client.providers.update(provider['id'], name='Jack')

# Refetch the provider record
```

```
provider = client.providers.get(provider['id'])

# Get providers with name containing 'Ja'
providers = client.providers.filter(name__like='Ja')

# Delete a provider record
client.providers.delete(provider['id'])
```

Provider Data Model

Providers currently store the following pieces of data [2]:

Name	Type	Description
id	String	Globally unique identifier
name	String	A readable name/identifier
location	String	Identifier for the provider's location
created_date	Datetime	Time when the record was created
updated_date	Datetime	Time when the record was last updated
status	String (A/I)	Flag to denote if the record is currently active

As with patients, the `location` field might store the name of the location or an identifier for another location/facility registry.

`providers.create`

`providers.create` adds a new provider record to the data store. The arguments for this function are passed to the backend to store on the record. The provider data is returned as a dictionary and contains additional fields which are generated by the backend: `id`, `created_date` and `updated_date`.

`providers.update`

`providers.update` takes the id of the provider along with arguments to be passed to the backend to update. This returns a boolean to note whether a matching provider was found and updated.

`providers.get`

`providers.get` returns a provider's data as dictionary for the given id. If no matching provider was found this will raise a `ProviderNotFound` exception.

`providers.filter`

`providers.filter` returns a list of matched provider data dictionaries. If there are no matches then it will be an empty list. Additional details on filtering expressions is given below.

`providers.delete`

`providers.delete` takes the id of the provider and returns a boolean to note whether a matching provider was found and deleted.

[2] Required fields are bold and generated values are in italics.

6.3.4 Filter Expressions

Both the patient and provider APIs support filtering the data by the fields in their respective models. The lookup expressions are modeled after the lookup types in the ORM. Unlike the Django ORM, there is no support for join-like expressions in the lookups.

`exact`

`exact` is the default lookup type. As the name implies it requires an exact match between the field and given value.:

```
patients = client.providers.filter(name='Joe')
providers = client.providers.filter(name__exact='Joe')
```

`like`

The `like` lookup is a containment expression for string-type fields. For instance, this would be used to find data with a partial name match.:

```
patients = client.providers.filter(name__like='J')
providers = client.providers.filter(name__like='J')
```

`in`

An `in` expression is an exact match for a list of values. This lookup might be used to find a set of patients where you know all of their names.:

```
patients = client.providers.filter(name__in=['Joe', 'Jane'])
providers = client.providers.filter(name__in=['Joe', 'Jane'])
```

`lt` and `lte`

Similar to the ORM, the `lt` and `lte` expressions are inequality expressions. These are used to find data either strictly less than or less than or equal to a given value respectively.:

```
import datetime

patients = client.providers.filter(updated_date__lt=datetime.datetime.now())
providers = client.providers.filter(updated_date__lte=datetime.datetime.now())
```

`gt` and `gte`

`gt` and `gte` expressions are inequality expressions. These are used to find data either strictly greater than or greater than or equal to a given value respectively.:

```
import datetime

patients = client.providers.filter(updated_date__lt=datetime.datetime.now())
providers = client.providers.filter(updated_date__lte=datetime.datetime.now())
```

6.4 Storage Backend API

If the existing storage backends do not met your needs then you can write you own backend. You may want to store data using a popular medical record system (OpenMRS, FreeMED) or NoSQL database (CouchDB, MongoDB). Since applications interact with the storages though the client API they will transparently work with your new storage.

Note: This is an advanced use case and not necessary for most users or application developers.

The data passed to the backend will match the data described in the *patient* and *provider* data model sections. Backend methods which return data should return dictionaries matching this format as well.

6.4.1 Backend API

Additional backends should extend from `HealthcareStorage` which is defined below:

class `HealthcareStorage`

`get_patient`(*id*, *source=None*)
Patient data should be fetched for the given `id` and returned as a dictionary. If the patient does not exist this method should return `None`.

`source` is an optional paramter. If given then the `id` should be interpreted as the `source_id` and the `source` as the `source_name` to find the patient using the association created by py:meth:*HealthcareStorage.link_patient*. If the patient cannot be found for this association it should also return `None`.

`create_patient`(*data*)
A patient record should be created for given set of `data` given as a dictionary. The newly created patient record should be returned as a dictionary.

`update_patient`(*id*, *data*)
Patient data for the given `id` should be updated with the `data` dictionary. `data` may not be a full set of the patient fields. This method should return `True` if a patient was found and updated and `False` otherwise.

`delete_patient`(*id*)
Patient data for the given `id` should be deleted. This method should return `True` if a patient was found and deleted and `False` otherwise.

`filter_patients`(**lookups*)
Returns a list of patients matching the set of lookups. If no patients were found it should return an empty list. If no lookups were passed it should return all patients. The details of the lookup structure is given in the next section. When multiple lookups are passed, the intersection of the results should be returned (default to AND the expressions).

`link_patient`(*id*, *source_id*, *source_name*)
Associates a patient with an addition identifier. The `source_id` and `source_name` pair should be enforced as unique. This should return a `True` value if the association was created. Otherwise it should return `False`.

`unlink_patient`(*id*, *source_id*, *source_name*)
Removes an association of a patient with an addition identifier. This should return a `True` value if the association was found and removed. Otherwise it should return `False`.

get_provider(*id*)
: Provider data should be fetched for the given `id` and returned as a dictionary. If the provider does not exist this method should return `None`.

create_provider(*data*)
: A provider record should be created for given set of `data` given as a dictionary. The newly created provider record should be returned as a dictionary.

update_provider(*id*, *data*)
: Provider data for the given `id` should be updated with the `data` dictionary. `data` may not be a full set of the provider fields. This method should return `True` if a provider was found and updated and `False` otherwise.

delete_provider(*id*)
: Provider data for the given `id` should be deleted. This method should return `True` if a provider was found and deleted and `False` otherwise.

filter_providers(**lookups*)
: Returns a list of providers matching the set of lookups. If no providers were found it should return an empty list. If no lookups were passed it should return all providers. The details of the lookup structure is given in the next section. When multiple lookups are passed, the intersection of the results should be returned (default to AND the expressions).

6.4.2 Backend Lookups

The above `HealthcareStorage.filter_patients()` and py:meth:*HealthcareStorage.filter_providers* methods are each passed a list of lookups for filtering the underlying records. Each of these lookups is a 3-tuple `(field_name, operator, value)`. The `field_name` is passed as a string and must match a field name on the corresponding data model. The `value` is the requested value for comparison which should be a standard Python type (int, float, list, sting, date, datetime, etc). The `operator` is one of the below constants from the `healthcare.backends.comparisons` module.

Operator	Comparison
`EQUAL`	Field is an exact match to the value
`LIKE`	Field contains the value
`IN`	Field is an exact match to one of the values in the value (list/tuple)
`LT`	Field is less than the value
`LTE`	Field is less than or equal to the value
`GT`	Field is greater than the value
`GTE`	Field is greater than or equal to the value

The backend is responsible for mapping these operators to the meaningful expressions for its storage method.

6.4.3 Testing the Backend

There is a testing mixin `BackendTestMixin` in `healthcare.tests.base` which runs through a set of compatibility tests for the backends. You simply need to attach the path of the backend to the `backend` attribute.

```
from django.test import TestCase

from healthcare.tests.base import BackendTestMixin


class FancyBackendTestCase(BackendTestMixin, TestCase):
    backend = 'path.to.new.backend'
```

This should not be considered an complete set of tests and the developers should write additional tests to cover edge cases in their backend.

6.5 Full Settings Reference

Below are the full set of settings for configuring rapidsms-healthcare along with their default values.

6.5.1 HEALTHCARE_STORAGE_BACKEND

Default: `'healthcare.backends.djhealth.DjangoStorage'`

Controls where rapidsms-healthcare stores patient and provider data. The default backends provided are:

- *healthcare.backends.dummy.DummyStorage*
- *healthcare.backends.djhealth.DjangoStorage*

Additional backends can be written as needed.

6.6 Contributing Guide

There are a number of ways to contribute to rapidsms-healthcare. If you are interested in making rapidsms-healthcare better then this guide will help you find a way to contribute.

6.6.1 Ways to Contribute

You can contribute to the project by submitting bug reports, feature requests or documentation updates through the Github issues.

6.6.2 Getting the Source

You can clone the repository from Github:

```
git clone git://github.com/caktus/rapidsms-healthcare.git
```

However this checkout will be read only. If you want to contribute code you should create a fork and clone your fork. You can then add the main repository as a remote:

```
git clone git@github.com:<your-username>/rapidsms-healthcare.git
git remote add upstream git://github.com/caktus/rapidsms-healthcare.git
git fetch upstream
```

6.6.3 Running the Tests

When making changes to the code, either fixing bugs or adding features, you'll want to run the tests to ensure that you have not broken any of the existing functionality. With the code checked out and Django installed you can run the tests via:

```
python setup.py test
```

or:

```
python runtests.py
```

To test against multiple versions of Django you can use install and use `tox>=1.4`. The `tox` command will run the tests against Django 1.3, 1.4 and the current Git master using Python 2.6.:

```
# Build all environments
tox
# Build a single environment
tox -e py26-1.3.X
```

Building all environments will also build the documentation. More on that in the next section.

6.6.4 Building the Documentation

This project aims to have a minimal core with hooks for customization. That makes documentation an important part of the project. Useful examples and notes on common use cases are a great way to contribute and improve the documentation.

The docs are written in ReST and built using Sphinx. As noted above you can use tox to build the documentation or you can build them on their own via:

```
tox -e docs
```

or:

```
make html
```

from inside the `docs/` directory.

6.6.5 Coding Standards

Code contributions should follow the PEP8 and Django contributing style standards. Please note that these are only guidelines. Overall code consistency and readability are more important than strict adherence to these guides.

6.6.6 Submitting a Pull Request

The easiest way to contribute code or documentation changes is through a pull request. For information on submitting a pull request you can read the Github help page https://help.github.com/articles/using-pull-requests.

Pull requests are a place for the code to be reviewed before it is merged. This review will go over the coding style as well as if it solves the problem indended and fits in the scope of the project. It may be a long discussion or it might just be a simple thank you.

Not necessarily every request will be merged but you should not take it personally if you change is not accepted. If you want to increase the chances of your change being incorporated then here are some tips.

- Address a known issue. Preference is given to a request that fixes a currently open issue.
- Include documentation and tests when appropriate. New features should be tested and documented. Bugfixes should include tests which demostrate the problem.
- Keep it simple. It's difficult to review a large block of code so try to keep the scope of the change small.

You should also feel free to ask for help writing tests or writing documentation if you aren't sure how to go about it.

6.7 Release History

Release and change history for rapidsms-healthcare

6.7.1 v0.1.0 (Released 2013-02-21)

- Initial public release
- Suport for Django ORM and in-memory backends