# rapidsms-decisiontree Documentation

***Release 0.1.0***

**Caktus Consulting Group, LLC**

**Jul 12, 2017**

# Contents

This application is a generic implementation of a decision tree, which is completely database-configurable. Users are asked questions and respond via SMS messages using the RapidSMS framework built on top of Django.

The original code for this application was written by Dimagi and is currently packaged and maintained by Caktus Consulting Group, LLC.

CHAPTER 1

# Requirements

*rapidsms-decisiontree-app* is tested on RapidSMS 0.19, Django 1.7, and Python 2.7. There is optional support for django-celery.

CHAPTER 2

# Features

- Support for sessions (i.e. 100 different users can all go through a session at the same time)
- Branching logic for the series of questions
- Tree visualization
- Errors for unrecognized messages (e.g. ‘i don’t recognize that kind of fruit’) and multiple retries before exiting the session

CHAPTER 3

# Installation

The latest stable release of rapidsms-decisiontree-app can be installed from the Python Package Index (PyPi) with pip:

```
pip install rapidsms-decisiontree-app
```

Once installed you should include `'decisiontree'` in your `INSTALLED_APPS` setting.

```
INSTALLED_APPS = (
    ...
    'decisiontree',
    ...
)
```

You'll need to create the necessary database tables:

```
python manage.py migrate decisiontree
```

At this point data can only be viewed/changed in the Django admin. If you want to enable this on the front-end you can include the `decisiontree.urls` in your root url patterns.

```
urlpatterns = [
    ...
    url(r'^surveys/', include('decisiontree.urls')),
    ...
]
```

See the full documentation for additional configuration options.

CHAPTER 4

# Running the Tests

Test requirements are listed in *requirements/tests.txt* file in the project source. These requirements are in addition to RapidSMS and its dependencies.

After you have installed `'decisiontree'` in your project, you can use the Django test runner to run tests against your installation:

```
python manage.py test decisiontree decisiontree.multitenancy
```

Minimal test settings are included in *decisiontree.tests.settings*; to use these settings, include the flag `--settings=decisiontree.tests.settings`.

To easily run tests against different environments that *rapidsms-decisiontree-app* supports, download the source and navigate to the *rapidsms-decisiontree-app* directory. From there, you can use tox to run tests against a specific environment:

```
tox -e python2.7-django1.7.X
```

Or omit the *-e* argument to run tests against all environments that *rapidsms-decisiontree-app* supports.

To see the test coverage you can run:

```
coverage run python manage.py test decisiontree decisiontree.multitenancy
coverage report -m
```

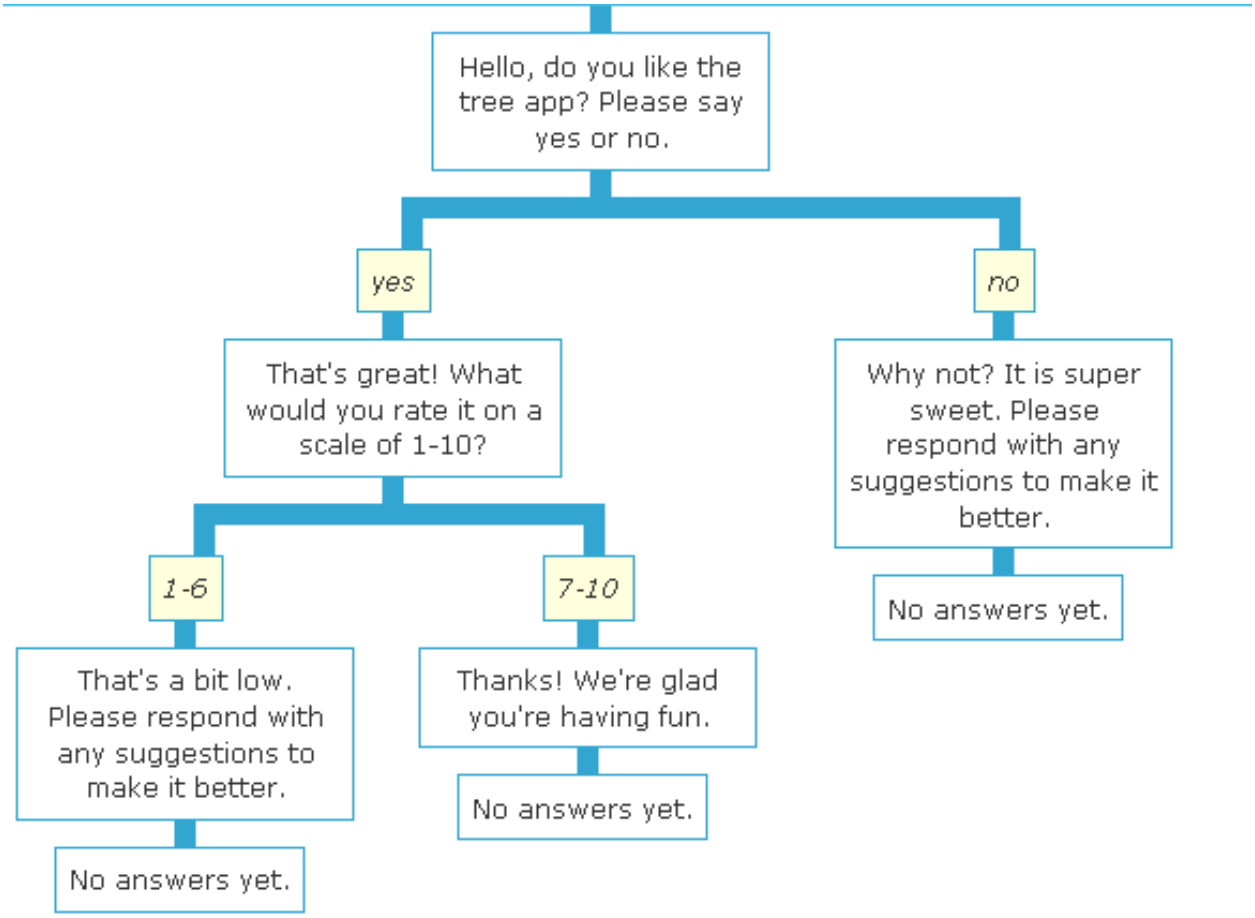
A common *.coveragerc* file is include in the repo.

CHAPTER 5

Contents:

## Decision Tree Overview

The tree app allows you to define decision trees that can perform a question-and-answer type interaction with a user. A tree consists of one or more states each of which is associated with a question and zero or more answers to that question that can transition to other states. As questions are answered the user traverses the tree based on the answers until he or she reaches a state that has no more transitions. At this point the user has completed the session. The tree saves every question/answer pairing in a single table, and provides functionality for applications to initiate a callback when trees are initiated and completed so that application developers can write their own processing of the tree data. A visualization of a tree is below.




## Making a Tree

Currently trees can only be made through the admin interface.

Trees have a trigger, which is is the incoming message that will initiate a tree. They also have a root state which is the first state the tree will be in. The question linked to the root state will be the one that is sent when the tree is initiated. The remaining logic of the tree is encapsulated by the Transition objects, which define how answers to questions move from one state to the next (more on this below).

A tree also has optional completion text, which is the message that will be sent to the user when they reach a node in the tree with no possible transitions.

## Questions/Answers/TreeStates/Transitions

The behavior of a tree is fully encapsulated a set of states and transitions that define how one moves through the tree.

A Question is just some text to be sent to the user, and an optional error message if the question is not answered properly.

A TreeState is a location in a tree. A TreeState is associated with a Question (that will be asked when the user reaches that state in the Tree) and a set of Answers (Transitions) that allow traversal to other TreeStates.

A Transition is a way to move from one TreeState to another. A Transition has a beginning state, an Answer, and an optional ending state. If a transition has no ending state, the answer will result in the completion of the tree.

An Answer is a way to answer a question, and defines how one moves across a Transition

There are three possible types of answers:

1. The simplest is an exact answer. Messages will only match this answer if the text is exactly the same as the answer specified.

2. The second is a regular expression. In this case the system will run a regular expression over the message and match the answer if the regular expression matches.

3. The final type is custom logic. In this case the answer should be a special keyword that the application developer defines. The application developer can then register a function tied to this keyword with the tree app and the tree app will call that function to see if the answer should match. The function should return any value that maps to True if the answer is valid, otherwise any value that maps to False.

## Registering a custom answer handler

The following code shows a function, and how to register that function with the Tree app as a custom answer handler for the word "demo".

```
# inside myapp.App

def validate_password(self, msg):
    """
    This function validates a password. This exact functionality could
    have been provided with a normal answer type, but you can put
    whatever logic you want here as long as you return True/False
    for matching/non-matching answers.
    """
    return msg.text == "spomc"

def start(self):
    """Start is called by the router to bootstrap our app"""
    # get the app from the rapidsms router
    self.tree_app = self.router.get_app("tree")
    # register our validate password function with the "demo" keyword
    self.tree_app.register_custom_transition("demo", self.validate_password)
```

## Notifications

For emailed notifications associated with tags to work, INSTALLED_APPS must contain 'rapidsms.contrib.scheduler' and the setting DECISIONTREE_NOTIFICATIONS must be True. The default is False.

## Timeouts

decisiontree can notice when it's been waiting for a response for too long and send a reminder, repeating the question. This requires running celery and celerybeat, and the additional configuration in settings.py.example.

On timeout, decisiontree will act as if it has received an invalid response. This results in sending a reminder and repeating the question, or, if the allowed retries are exhausted, giving up.

# Simple Tree Example

Below is the setup for an example survey which asks three basic questions to a user. There are no branches and all responses allow for free text. The questions are based on the bridgekeeper scene from "Monty Python and the Holy Grail".

## Creating Questions

We will create the three questions via the admin interface with the following data:

```
# Questions
pk: 1
text: What is your name?

pk: 2
text: What is your quest?

pk: 3
text: What is your favorite color?
```

**Note:** The pks are listed for future reference but these would be auto generated by the database.

## Creating Answers

Next we will create an allowable answer in the admin. These questions don't require exact matches so we will just accept any text:

```
# Answers
pk: 1
name: Free Text
type: Regular Expression
answer: .*
```

**Note:** This regular expression will match anything. You may need to make this express less allowing depending on your needs.

## Associating Questions and Answers

Questions and answers are associated via tree states and transitions. Since we don't have any branches the tree states and questions will be tied in a one to one fashion:

```
# Tree States
pk: 1
question: 1

pk: 2
question: 2
```

```
pk: 3
question: 3
```

Within a tree state you can also optionally specify the number of allowable retries but these have been excluded for simplicity.

Transitions determine how users are moved through the tree based on their answers. Here we will allow any text for each question and simply move the user on to the next question:

```
# Transitions
pk: 1
current state: 1
answer: 1
next state: 2

pk: 2
current state: 2
answer: 1
next state: 3

pk: 3
current state: 3
answer: 1
next state: null
```

Transitions can also be tagged and notifications can be sent when those tags are triggered. For instance you may create a new answer which is for the text 'blue' and have a new transition which handles the case when the user had the favorite color blue. Again this has been excluded for simplicity.

## The Survey Tree

At this point the tree structure is in place to ask the series of questions but we need a way to start the survey. Again in the admin we would create a Tree with a trigger keyword which asks the user the first question:

```
# Tree
pk: 1
trigger: #test
root state: 1
completion text: Go on. Off you go.
```

Now when an incoming SMS matches the trigger text `#test` we will respond with the question from state one "What is your name?". They will proceed on with each question and when finished we will respond "Go on. Off you go." This completion text is optional.

At this point we have a simple yet functioning linear survey tree. An example SMS workflow is given below:

```
555-555-1234 >>> #test
555-555-1234 <<< What is your name? # This is state 1, question 1
555-555-1234 >>> My name is Sir Lancelot of Camelot.
555-555-1234 <<< What is your quest? # This is state 2, question 2
555-555-1234 >>> To seek the Holy Grail.
555-555-1234 <<< What is your favorite color? # This is state 3, question 3
555-555-1234 >>> Blue.
555-555-1234 <<< Go on. Off you go. # End of questions
```

Continuing reading to see how we can add branches to this series of questions.

# Advanced Tree Example

Here we will expand on the previous tree example to contain branches based on varying answers. We will continue to work from the Monty Python questions.

## New Questions

First will add new questions for the addition branches:

```
# Questions
pk: 4
text: What is the capital of Assyria?

pk: 5
text: What is the air-speed velocity of an unladen swallow?
```

## New Answers

In the same logic of the movie we will ask the questions based on the name of the knight:

```
# Answers
pk: 2
name: Sir Robin
type: Regular Expression
answer: .*Robin.*

pk: 3
name: King Arthur
type: Regular Expression
answer: .*Arthur.*
```

## New States and Transitions

We need additional states to handle these trees. Since the final question is the only thing different we still need multiple states which point to the same second question:

```
# Tree States
pk: 4
question: 4

pk: 5
question: 5

pk: 6
question: 2

pk: 7
question: 2
```

Now we need the transitions between the questions based on the first answer. First we will do Sir Robin:

```
# Transitions
pk: 3
current state: 1
answer: 2
next state: 6

pk: 4
current state: 6
answer: 1
next state: 4

pk: 5
current state: 4
answer: 1
next state: null
```

Then King Arthur:

```
# Transitions
pk: 6
current state: 1
answer: 3
next state: 7

pk: 7
current state: 7
answer: 1
next state: 5

pk: 8
current state: 5
answer: 1
next state: null
```

The tree itself does not need to be modified. An example SMS workflow is given below:

```
555-555-1234 >>> #test
555-555-1234 <<< What is your name? # This is state 1, question 1
555-555-1234 >>> Sir Robin
555-555-1234 <<< What is your quest? # This is state 6, question 2
555-555-1234 >>> To seek the Holy Grail.
555-555-1234 <<< What is the capital of Assyria? # This is state 4, question 4
555-555-1234 >>> I don't know that.
555-555-1234 <<< Go on. Off you go. # End of questions
```

So this isn't perfect to movie but it should demonstrate the difference from the simple example.

## Available Settings

rapidsms-decisiontree-app has a few settings available for configuring the behaviour.

### DECISIONTREE_NOTIFICATIONS

Default: `False`

If enabled this will periodically send emails based on the response tags and the `TagNotification` configurations. This requires the `rapidsms.contrib.scheduler` app.

### DECISIONTREE_SESSION_END_TRIGGER

Default: `end`

This configures a keyword which the users can use to end their question session. This functionality can be disabled by making this setting `None`.

### DECISIONTREE_TIMEOUT

Default: `300`

This is the time in seconds to wait between questions before the user is asked the question again or the question session is abandoned. Using this setting requires the threadless-router and django-celery. You must enable this task in your `CELERYBEAT_SCHEDULE` in your project settings

```
from celery.schedules import crontab

CELERYBEAT_SCHEDULE = {
    # Other periodic tasks included here
    "decisiontree-tick": {
        "task": "decisiontree.tasks.PeriodicTask",
        # How often to check sessions for timeout
        "schedule": crontab(),  # every minute
    },
}
```

## Release History

Below is the history of the rapidsms-decisiontree project. With each release we note new features, large bug fixes and any backwards incompatible changes.

### v0.1.0 (TBD)

The initial PyPi release.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search