Forms System Renderer Extension

Introduction

This system uses Form System XML and connects to the Forms System Questionnaire database to provide extra functionality for the system.

It uses the Django Web Framework to provide this functionality. The Questionnaire models are provided by the django-fsq package and the xml objects by the XML Objectifier package.

Features

It provides...

- builtin rendering of Forms System XML
- ability to extend functionality using a plugin architecture.

History

The Forms System Renderer Extension is a stripped down version of the oirginal project stripped down to isolate the functionality solely suited for the use of the Questionnaire models.

Installation

Set up a local virtualenv in a selected directory.

```
virtualenv .
```

Activate the environment

```
Scripts\activate
```

Download or clone the files from the relevant project from the repository e.g.

```
git clone https://github.com/cam-mrc-epid/fs_api2
```

Note it is also available to checkout via SVN at
https://svn.mrc-epid.cam.ac.uk/private/MRC_Webforms_Python/fs_api2

Go into the Git repo root, e.g.

```
cd fs_api2
```

Now install the project requirements

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

Note in some cases you may receive admin required warnings on Windows, just click `cancel` if they come up. Some modules want to install C modules for accelerating things but provide standard backups if they aren't installed.

Most of the requirements will be installed in your virtualenv's lib/site-packages directory. If any of the requirements are installed from git repos they may be installed in a src directory in your virtualenv's root beside the lib directory.

If you need to make changes to installed code it is best to fork the code yourself, don't edit it in the lib folder. The src folder contain git checkouts of the installed app code and you can manage these apps from this place as they are attached to git but you will likely have to make a few changes before git allows it.

Usage

This project is meant to be used via the Forms System e.g. sections, question groups or questions within the XML may instead be designated to pass a request to Django. This project will accept get or post requests and will return a chunk of html. URLs have to be in the following forms:

To get down to a question or textnode level:

```
http://server-name:port/althtml/<section>/<question
group>/<question>/?id=<username>
```

Where `<section>`, `<question group>` and `<question>` are the position numbers of the elements, e.g.

```
http://server-name:port/althtml/1/2/3/?id=davidg
```

This would return the html for question in position 3, in Question Group 2 that is in Section 1. Likewise

```
http://server-name:port/althtml/<section>/<question group>/?id=<username>
```

This will return the full Question Group html, e.g.

```
http://server-name:port/althtml/1/2/?id=davidg
```

would return the full html for question group 2 in Section 1. For the full section html it needs to be

```
http://server-name:port/althtml/<section>/?id=<username>
```

e.g.

```
http://server-name:port/althtml/1/?id=davidg
```

This would return Section 1.

Configuration

There is a limited number of configuration options in fs_renderer/local_settings.py

Plugins

PLUGINS is a dictionary matching plugin name to a plugin class.

TESTING is a boolean which if set to True will add HTML header and footer to the templates along aith a submit button for testing. This is set for each section object and can be used in the templates to show html you would only like to show in testing. This is done in the following way:

```
    {% if section.testing %}
        <p>Some additional code, can be anything, javascript, styling etc.</p>
    {% endif %}
```

CUSTOM is set to False by deafult but if set to True will use the CustomApplication object in fs_renderer/custom_logic.py. Recently a new requirement to possibly load multiple XML files will require a change in this style. Application objects can be loaded per request for different XML files or instantiated at startup and reused. This is not set up by deafult.

XML_FILE is set to point to Questionnaire XML file as the application objects instantiated in the views depend on it. The new requirement of having to load multiple XML files may make it redundant.

Extending the Renderer

There are a number of ways of extending the functionality of the renderer.

Subclassing

You can subclass the Application class (fs_renderer/fs_apps.py) and override functionality there. There is an example file in the app (fs_renderer/custom_logic.py). The initial development of this expected a single XML file per app so a custom class could be written and used for that app and the CUSTOM variable set to True (fs_renderer/local_settings.py). This is set to False by default.

More recent requirements suggested that multiple XML files may be used in a single app. Therefore it may be necessary to change this approach by having a custom class per XML file and loading them in the views as necessary. For custom functionality it may be easier to use the plugins architecture.

To use the custom_logic modules CustomApplication class you need to set CUSTOM to True in fs_renderer/local_settings.py.

Plugins

Plugins arose from the need to have extra functionality in the section interfaces that required the interface to build an unknown number of rows etc. or any more complicated interface that would be difficult to describe in the XML.

They can be set up as follows:

In the XML you can replace a section, question group or question with a plugin by using the rendering hint:

```
                <epi:renderingHint>
                    <epi:rhType>plugin</epi:rhType>
                    <epi:rhData>some_plugin</epi:rhData>
                </epi:renderingHint>
```

where the `some_plugin` is the name of your plugin.

Now register the plugin in fs_renderer/local_settings.py

```
PLUGINS = {'childlist': ChildList, 'some_plugin': SomePlugin}
```

where the key 'some_plugin' is the name you used in the XML and SomePlugin is a class you will create in fs_renderer/plugins.py.

In fs_renderer/plugins.py you build a plugin class e.g.

```
class SomePlugin():
    def __init__(self, data, caller):
        self.data = data
        self.caller = caller #
        if isinstance(caller,fs_apps.Question):
            self.template = 'fs_renderer/some_plugin_q.html'
```

```
        else:
            self.template = 'fs_renderer/some_plugin.html'

    def do_it(self):
        if isinstance(self.caller,fs_apps.Question):
            results = Results.objects.filter(var_name__startswith='child_age')
            tally = 0
            number = 0
            for result in results:
                if result.var_value != '':
                    tally = tally + int(result.var_value)
                    number = number + 1
            average = tally/number
            return average
        if isinstance(self.caller,fs_apps.QuestionGroup):
            profile = {}
            profile['name'] = self.caller.title
            profile['type'] = 'QuestionGroup'
            profile['position'] = self.caller.position
            profile['section'] = self.caller.section.title
            return profile
```

In this example the plugin class SomePlugin accepts two arguments data and caller. data is a dictionary pulled from the questionnaire database relating to the participant and their data for the questionnaire they are completing. The caller is the section, question group or question that the plugin will replace. This example will return different things based on the type of caller it receives.

A plugin must set a template that will be used to render the plugin. The plugin instance will be passed to the template so the template can access its properties and methods directly.

An example template in this case would be fs_renderer/templates/fs_renderer/some_plugin.html:

```
<div>
    <ul>
        <li>{{ question_group.plugin.do_it.name }}</li>
        <li>{{ question_group.plugin.do_it.type }}</li>
        <li>{{ question_group.plugin.do_it.position }}</li>
        <li>{{ question_group.plugin.do_it.section }}</li>
    </ul>
</div>
```

This plugin is for question groups and must use question_group.plugin to run its plugin methods. Likewise it would be question.plugin for a question and section.plugin for a section.

You can import models from any app to use in the plugins and therefore can access different databases via their models. You can use any python module available to do processing.

Project Structure

Within the repository root the **fs_proj** folder is the Django project root.

fs_proj/fs_proj is the Django project settings folder containing settings and urls files.

fs_proj/fs_proj/settings.py is a standard settings file. It contains a sample database connection for a MySQL database with the Forms System setup. The key 'db3' is used by the **questionnaire** app's routers to connect to the MySQL database. You need to change the settings to reflect your setup. There is also a DATABASE_ROUTERS setting which points to the **questionnaire** app's router class (see Installation section).

fs_proj/fs_proj/urls.py has the project urls. They simply use `/althtml/<section>/<question_group>/<question>/` style. The original setup used /html hence /althtml was a later addition. The urls, with the exception of the admin one all use the same view and pass the parts of the url as positional arguments to the same view class.

fs_proj/fs_renderer is the Django app. The app is different from the standard Django app structure in that it imports the models from the *questionnaire* which is installed by the Django-FSQ package. The questionnaire app provides its own admin screens for its models and also routers for the database (See Django FSQ docs.).

fs_proj/fs_renderer/fs_apps.py contains the Application objects that reflect the XML file. It is imported in the views.py file.

fs_proj/fs_renderer/local_settings.py contains the project settings (See configuration section).

fs_proj/fs_renderer/plugins.py contains some sample plugin classes and is the location for any plugin classes you need.

fs_proj/fs_renderer/views.py is a normal Django style views file using class based views. These separate out GET and POST requests to use the class get() and post() methods.

fs_proj/fs_renderer/templates conatins the app html templates. These follow an include pattern as was necessary to allow question html to rendered on its own.

fs_proj/xmlfiles contains some example XML files.

Project Dependencies

arrow==0.6.0 # used in custom_logic example

bunch==1.0.1 # installed by XML Objectifier

diff-match-patch==20121119 # django-import-export dependency

Django==1.7.5

django-extensions==1.5.7 # debugging extension, can be removed for production.

-e

git://github.com/davidgillies/django-fsq@cca77b4599366953fc523bf4f58acbd35d052067#egg=django_fsq-master # specific commit of the Django FSQ app.

django-import-export==0.2.8 # dependency of the Django FSQ app.

lxml==3.4.4 # dependency of the XML Objectifier

PyMySQL==0.6.6 # Required to connect to MySQL database

python-dateutil==2.4.2 # dependency of arrow

six==1.9.0 # dependency of django-extensions, can be removed for production

tablib==0.10.0 # dependency of django-import-export -e git://github.com/davidgillies/xml-objectifier@dd41d57dfbb57b9bad85a0e6f110f3837508b3bd#egg=xml_objectifier-master # used to provide inital Application models, subclassed is fs_renderer/fs_apps.py