ServiceInfo Documentation

Release 0.5.3

International Rescue Committee

Sep 21, 2017

Contents

The International Rescue Committee (IRC), one of the world's leading humanitarian organizations providing relief and resettlement services to people impacted by crisis, created ServiceInfo, an online platform that enables the more than 1 million Syrian refugees in Lebanon to search and rate aid and commercial services – ranging from healthcare to financial services and retail – online or over the phone.

Developed by the IRC and funded by a grant from the U.S. Bureau of Population, Refugees and Migration, the IRC's ServiceInfo project aims to provide a solution to an information gap for transient refugees looking for local services.

This is ServiceInfo's user and developer documentation. To explore other open source IRC projects, please see IRC's GitHub account.

Contents:

CHAPTER 1

Development Setup

Clone the Repository

To get started, you'll first need to clone the GitHub repository so you can work on the project locally. In a terminal, run:

```
git clone git@github.com:theirc/serviceinfo.git
cd serviceinfo/
```

Frontend Setup

The frontend runs separately from the backend.

The Javascript dependencies are installed by Node's NPM, both for build tools and frontend modules. Javascript libraries for the frontend app are installed from NPM and then packaged for the browser by Browserify. You'll need to install Node, which includes npm, in order to build the frontend application if you'd like to run it.

On Mac, you can install Node with brew:

```
brew install node
```

On Ubuntu and other Linux distributions, you should download and build the latest version of Node v0.10.*. (Newer versions might not work.)

Standard package managers rarely have the most recent versions of Node that include NPM. You can download it from http://nodejs.org/download/ and follow the standard build instructions:

```
wget https://nodejs.org/download/release/latest-v0.10.x/node-v0.10.40.tar.gz
tar -zxf node-v0.10.40.tar.gz
cd node-v0.10.40/
./configure
make
```

```
sudo make install
cd ..
```

WARNING: Do not build node while your ServiceInfo virtualenv is active. Node expects Python 2.7 and our virtualenv uses Python 3.

With Node installed, you can install all frontend dependencies with *npm*:

```
npm install
```

Backend Setup

Below you will find basic setup instructions for the project. To begin you should have the following applications installed on your local development system:

- Python == 3
- pip >= 6.1.0
- virtualenv >= 1.11
- virtualenvwrapper >= 3.0
- Postgres >= 9.1 (9.5 recommended)
- PostGIS
- git >= 1.7
- node
- npm
- Elasticsearch 1.7.4

The deployment uses SSH with agent forwarding so you'll need to enable agent forwarding if it is not already by adding `ForwardAgent yes` to your SSH config.

Getting Started

If you need Python 3 installed on Ubuntu, you can use this PPA:

```
sudo add-apt-repository ppa:fkrull/deadsnakes
sudo apt-get update
sudo apt-get install python3-dev
```

To setup your local environment you should create a virtualenv and install the necessary requirements:

```
mkvirtualenv --python=<path>/python3 serviceinfo
pip install -U pip
pip install -U -r requirements/dev.txt
```

In order to use JavaScript tools that npm installs, you'll need to add `node_modules/.bin` to the *front* of your PATH. One way to do that is to add this to your virtualenvs' postactivate script:

```
if [ -d node_modules/.bin ] ; then
    if printenv PATH | grep --quiet node_modules/.bin ; then
        echo "node_modules already on PATH: $PATH"
    else
        echo "Adding node_modules to PATH"
        PATH=$(pwd)/node_modules/.bin:$PATH
    fi
fi
```

Install the needed Javascript tools and libraries:

```
npm install
```

Then create a local settings file and set your `DJANGO_SETTINGS_MODULE` to use it and also to use the javascript tools just installed:

```
cp service_info/settings/local.example.py service_info/settings/local.py
echo "export DJANGO_SETTINGS_MODULE=service_info.settings.local" >> $VIRTUAL_ENV/bin/
↪postactivate
echo "unset DJANGO_SETTINGS_MODULE" >> $VIRTUAL_ENV/bin/postdeactivate
echo "PATH=$PWD/node_modules/.bin:\$PATH" >> $VIRTUAL_ENV/bin/postactivate
```

Exit the virtualenv and reactivate it to activate the settings just changed:

```
deactivate
workon serviceinfo
```

Now you can run the tests:

```
./run_tests.sh
```

Enabling the search engine

Running Elasticsearch can be as simple as unpacking it and then:

```
cd elasticsearch-1.7.4 && bin/elasticsearch
```

(This requires Java.)

You should add this to the bottom of `config/elasticsearch.yml` to limit it to a simple single-node configuration which only services the local machine:

```
network.host: 127.0.0.1
node.local: true
discovery.zen.ping.multicast.enabled: false
```

If you have less than 10% disk space free, you'll need to make more space available or add this to the bottom of `config/elasticsearch.yml`:

```
cluster.routing.allocation.disk.threshold_enabled: false
```

Use the Django management commands `rebuild_index`, `clear_index`, or `update_index` to maintain the search index. (The index will be updated in real time after some types of changes.)

Disabling search indexing

Add this to `local.py`:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.BaseSignalProcessor'
```

Running locally

Create the Postgres database and run the initial migrate:

```
createdb -E UTF-8 service_info
psql service_info -c "CREATE EXTENSION postgis;"
python manage.py migrate
```

You should now be able to build the frontend and run the development API server:

```
gulp
```

Follow the instructions for CMS configuration in the CMS setup document or just run the `create_minimal_cms` management command.

Now visit http://localhost:4005/ in your browser.

If you need to debug the Javascript, you might prefer to skip running Closure. You can skip closure by adding the `--fast` option to gulp:

```
gulp --fast
```

Celery

Use this to run a single worker with the "beat" task scheduler:

```
celery -B -A service_info worker -l debug
```

Using the staging or production database and media locally

Changes relating to the CMS, such as those affecting page templates and styles or CMS plugins, should be tested locally with the staging and/or production databases and media in order to check how the existing content will be affected. The procedure uses commands in both the ServiceInfo and ServiceInfo-ircdeploy repositories:

```
$ cd ServiceInfo-ircdeploy
$ workon virtualenv-with-fab
$ fab production reset_local_db
$ fab production reset_local_media:../ServiceInfo
$ cd ../ServiceInfo
$ workon virtualenv-for-ServiceInfo
$ ./manage.py migrate
$ ./manage.py change_cms_site --from=serviceinfo.rescue.org --to=localhost:8000
# If using search locally
$ ./manage.py rebuild_index --noinput
```

CHAPTER 2

API

For the most part, the API uses the defaults of Django REST Framework to provide access to the models in the usual way. You can browse the API at *https://<yourserver>/api*. This document will only cover the little complications, like getting authenticated, creating users, password reset, etc.

Ownership

The API enforces ownership in a few places. When creating records that are owned by a particular provider, the provider is forced to be that of the currently authenticated user. Similarly, when querying records of those models, only those owned by the currently authenticated user are returned. The relevant record types are Service and SelectionCriterion for creation, and those plus Provider for querying.

Paginating results

By default the API returns all results in a single response. The caller can optionally provide *limit* and *offset* fields to retrieve a subset of the results using DRF's LimitOffsetPagination.

Creating a new provider

The usual way to create a new instance of a model would be to POST to the model's list URL. However, for new provider registration, it's necessary to be able to create a new provider when not authenticated, and the usual APIs don't allow that. We also want to create both a user and a provider record. So we've created a custom call just for this.

To use it, POST to `/api/provider/create_provider/`. The request data is almost the same as it would be to create a Provider normally, except instead of a 'user' field, it expects 'email' and 'password' fields, plus a 'base_activation_link' field:

```
{'email': 'user's email address',
 'password': 'plaintext password',
 'base_activation_link': 'used to build activation link',
 # remaining fields are for Provider, omitting 'user' field
}
```

The 'email' and 'password' fields ought to be self-explanatory.

The 'base_activation_link' should be a URL that the activation key can be appended to, giving the link the user should be sent to activate their account. E.g. base_activation_link might be "https://example.com/activate?key=" and then the API would append the actual key string ("XYZ") and send the result ("https://example.com/activate?key=XYZ") in the email to the user. The resulting link should invoke the client, which should then use the user activation API (see below) with the given key.

When the create_provider call is successful, a new, inactive user will be created, and an email sent to the user with the activation link.

Also, a new provider will be created for that user.

200 is returned and the response body is the response from creating a provider, for example:

```
{'description_ar': '',
 'description_en': 'Test provider',
 'description_fr': '',
 'id': 2,
 'name_ar': '',
 'name_en': 'Joe Provider',
 'name_fr': '',
 'number_of_monthly_beneficiaries': 37,
 'phone_number': '12345',
 'type': 'http://testserver/api/providertypes/2/',
 'url': 'http://testserver/api/providers/2/',
 'user': 'http://testserver/api/users/16/',
 'focal_point_name_en': 'John Doe',
 'focal_point_name_ar': '',
 'focal_point_name_fr': '',
 'focal_point_phone_number': '87-654321',
 'address_fr': '1 Rue Madeleine, Paris',
 'address_en': '',
 'address_ar': '',
 'website': ''}
```

If there's a problem, 400 is returned and the response body might be something like one of these:

```
{'email': ['Enter a valid email address.']}
{'email': ['This field may not be blank.']}
{'password': ['This field may not be blank.']}
```

Fetch a Provider anonymously

The ordinary API for getting providers requires an authenticated user, and only returns the provider owned by that user, but includes all information and allows updates.

For use in the search and display of services, there's a an alternate API to fetch the public data for a specific provider without requiring authentication. To use it, `GET /api/providers/NNN/fetch/` where NNN is the ID of the provider. Service results have also been augmented with a `provider_fetch_url` field that will provide that URL for use in this API.

Resend activation link

If a user has lost their activation link, the client can POST to `/api/lost_activation_link/`:

```
{'email': 'user's email address',
 'base_activation_link': 'used to build activation link',
}
```

and if there's a account under that email waiting to be activated, an email will be sent to that email containing an activation link and a 200 returned. Otherwise, a 400 status and error message will come back. FILL ME IN.

User activation

If the client has a user activation key, the client can try to activate the user by POSTing to `/api/activate/` with

{ 'activation_key': 'the key string' }

If successful, response status will be 200 and the response content will include:

```
{ 'token': 'a long string',
  'email': 'the user's email address'}
```

The token can be used to make subsequent API calls with the permissions of that user (see below).

Otherwise, the response status will be 400 and the body might contain:

```
{'activation_key': ['Activation key is invalid. Check that it was copied correctly '
                    'and has not already been used.']}
{'activation_key': ['This field may not be blank.']}
```

Login

If a client has a user's email and password, it can get an auth token and use that in subsequent calls.

POST to '/api/login/':

```
{ 'email': 'email@example.com',
  'password': 'plaintext password' }
```

If successful, response status will be 200 and the response content will include:

```
{ 'token': 'a long string',
  'language': 'xx', # User's preferred language code, e.g. 'en' or 'ar',
                    # or '' if we don't know
  'is_staff': true or false    # whether this is a staff user (allowed to use Django␣
↪admin)
}
```

If failed, response status will be 400 and the response might look like one of these:

```
{"non_field_errors": ["Unable to log in with provided credentials."]}
{"email": ["This field may not be blank."]}
{"non_field_errors": ["User account is disabled."]}
```

Using token-based auth

Once the client has the token, it should pass it on subsequent requests, including it in the ServiceInfoAuthorization HTTP header, prefixed by the string literal "Token" with whitespace between:

```
ServiceInfoAuthorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

As you might expect, requests will be permitted or denied based on the permissions of the user whose token is passed.

User language

The client can store and retrieve a short string containing the code for the current user's preferred language:

```
GET /api/language/
  --> {'language': 'en'}

POST {'language': 'en'} to /api/language/
```

At present, the language code should be one of

- "en": English
- "ar": Arabic
- "fr": French

Password reset

If a user wants to reset their password, the client should POST to `/api/password_reset_request/`:

```
{'email': 'user@example.com',
 'base_reset_link': 'https://example.com/reset?key=',
}
```

where email is the user's email. The server will generate a new key (a long string that the client should not try to interpret), specific for this user to reset their email, append it to the base_reset_link, and email it to the given email address, then return 200. Or if there's no such user or other error, return 400 and an error message.

The front end should arrange to handle the resulting URL. Ask the user for a new password. Then POST to `/api/password_reset/`:

```
{'key': 'the password reset key',
 'password': 'the new password'}
```

If the response status is OK (200), then the body will have

{'email': 'the user's email address', 'token': 'a valid auth token for the user'}

Otherwise the status will be 400 and the body will have error messages. The reset can fail because the key is missing, has the wrong syntax, is not recognized, has already been used, has expired, etc:

```
{"email": ["No user with that email"]}
{"non_field_errors": ["Password reset key is not valid"]}
```

If the front end wants to check if the password reset key looks like it's probably valid before prompting the user for a new password, it can optionally POST to `/api/password_reset_check/`:

```
{'key': 'the password reset key'}
```

and will get back OK if the key appears to be valid, and the associated email address in the response:

```
{'email': 'user@example.com'},
```

Otherwise,it'll get a 400 but no other data.

Editing a Service

Users of the API may NOT modify existing service records. They need to create a new Service and set update_of to the previous record, which will kick off a backend process where a human will review the changes and switch the new service to being the active one if they approve.

Clients may submit an edit of a record that is a pending change to a current record. Just create yet another new record and set update_of to the draft record they're updating.

When that happens, though, the previous draft record will be archived, essentially making it go away, and only the most recent submitted update record will be visible in most places.

Cancel a Service

A provider can cancel a current service to withdraw it from the directory, or cancel a service record that is in draft status to cancel the requested new service or change.

The URL for this API is the service's URL with 'cancel/' appended. POST to it to do the cancel.

On success it'll return a 200. If the service isn't in a valid state to be canceled, it'll return 400. If the service doesn't belong to the provider making the call, it'll return a 404 (because only services belonging to a provider are visible to the provider).

Searching services

There's a separate call for searching public information about services without regard to the user's login status or permissions. The URL is `/api/services/search/`.

Service filtering

On both the normal list API and the search API for service, filtering is available.

For example, appending `?name=foo` will return services whose name in any language, case insensitively, contains "foo".

List of filter queries for services:

- name = service name in any language
- area_of_service_name = name of area of service in any language

- description = description (in any language)
- additional_info = additional info (in any language)
- type_name = the name of the service type desired (in any language)
- type_numbers = a comma-separated list of the numbers of the service types to include
- id = a specific service's id (primary key)

Full-text search

Additionally full-text search is available. Append `?search=XYZ` to either the list or search URLs to search for services with XYZ in pretty much any of the text fields associated with the service, its provider, its type, or its service area.

Searches will use case-insensitive partial matches. The search parameter may contain multiple search terms, which should be whitespace and/or comma separated. If multiple search terms are used then objects will be returned in the list only if all the provided terms are matched.

Sort by distance

When listing or searching services, if a `closest` query param is provided containing a comma-separated latitude and longitude (in that order), then the results will be sorted with the items closest to the specified point first. E.g. `?closest=35.5,-80`.

If results are sorted by distance, then each result will have a `distance` field set to the number of meters the result is from the given point. Otherwise, the distance will just be zero and should be ignored.

Latitude is north-south position, positive meaning north of the equator. Longitude is east-west position, positive meaning east of Greenwich and negative meaning west. Units are degrees, expressed in decimal.

North Carolina is at 35.5000° N, 80.0000° W, which we would pass to this API as "35.5,-80" meaning 35.5 degrees north latitude and 80 degrees west longitude.

Data export

A user can download an Excel spreadsheet with data. To do so via the API, login, then call */api/export/*. The return value will contain a *url* value. That is a time-limited signed URL that can be used to download the spreadsheet. It will expire in a few minutes, and can only be used for exporting data for this user.

Data import

To import a spreadsheet of data in the same format, use the */api/import/* API and submit *multipart/form-data* containing an uploaded file named *file* with the spreadsheet.

CHAPTER 3

CMS Setup

This document describes necessary page setup for the site to function properly.

When adding a language

- Create variation of search-results page for that language and publish.

Search bar support

- Create a page to display search results. On Advanced Settings page:
  1. Id must be set to “search-results”
  2. Application must be set to “aldryn search”
  3. In the Pages menu, un-check the Menu column.
  4. For each language:
  - Set title to “Search Results” (or equivalent) in the appropriate language
  - Publish.

  The `create_minimal_cms` management command can create the page and initial language variations. After running the command, edit the page titles and other data as appropriate.

CHAPTER 4

Import/Export

There is be an “Import/Export” item on the menu that goes to an import/export page.

Downloaded and uploaded Excel files will have the same format, as follows:

- The spreadsheet file will have three sheets or tabs.
- The first will contain the provider’s information, as a header row of field names followed by a row with one field’s data in each column.
- The second sheet will have the data for the provider’s currently approved/public services, again with a header row of field names, followed by one row per service with a field in each column.
- The third sheet will have the selection criteria for the services.

Pending new services or changes will not be included in the exported data.

Each sheet will have a first column called “id” which is a number that identifies that provider or service internally and is used when importing to match up the data to an existing record.

If an imported file contains services with blank ‘id’ fields, those rows will be considered as requests to create new services.

Rows with valid ID fields will be considered requests to change the data for existing services. If changes are already pending to the existing service, the pending changes will be canceled and replaced by the new requested changes.

Any rows with ID fields that are non-existent services or not currently public services belonging to the provider will cause the entire import to be rejected and the errors reported to the user.

Any row with a valid ID field but with all the other values cleared out will signal a request to delete/cancel that record.

The Services sheet will have a second column “provider__id” with the ID of the provider who owns the service. Providers should just leave this column alone when importing, and copy its value to any new service they are requesting to create.

The selection criteria sheet will have these columns: ‘id’, ‘service__id’, ‘text_en’, ‘text_ar’, and ‘text_fr’. ‘id’ is a unique identifier for that criterion. ‘service__id’ is the ID of the service it applies to. The other fields should be self-explanatory.

Times of day will be represented as a string in 24-hour “HH:MM” format.

If the data for the provider is edited in the first sheet and is valid, then the provider's data will be updated with the new data upon successful import.

The provider sheet will have a column "password" that never contains any values on export. The provider password can be changed by putting a value in that column for import.

Some fields in the provider and service records in the database do not contain values directly, but are links to other tables. The export/import format will not try to represent those fields that way. Any field that in the database is a link to a record in another table will be represented in the spreadsheet by one or more fields containing the data in the record that was linked to. For example, the provider type field links to a provider type record, but when the provider data is exported, instead of a single type column there will be four columns: type__number, type__name_en, type__name_ar, and type__name_fr (these are the four columns in the provider type table).

When adding selection criteria to a new service, there's no way to know what ID the new service will be assigned. So in the selection criterion's 'service__id' field, instead put the english name of the new service, for lack of a better solution.

When editing the spreadsheet to change the data, the user must either not change the values in these columns or must copy exactly the data from another record in the other table. On import, if e.g. type__number, type__name_en, type__name_ar, and type__name_fr do not match exactly the four values in one record in the type table, then the import will be considered invalid and not accepted. Errors will be reported to the user.

When importing a spreadsheet, all data will be carefully checked and if anything does not appear valid, the entire import will be rejected.

Staff export/import

If a user is flagged as a staff user in Django (e.g. IRC staff), then the export button on the services list page will download a spreadsheet containing the data for all providers. (As if all providers downloaded their data and the results were merged into a single file). The format is exactly the same.

Import will work similarly to provider import, with these changes:

- New providers can be created by including their data on the providers sheet with a blank ID, including a new password in the password column. New providers will be sent the usual confirmation email with a link they have to follow before they can login.
- New services can be created by including their data on the services sheet with a blank ID. If the provider who owns the service already exists, put their ID in the Provider_ID column. If the provider is being created by this import, put the new provider's "name_en" value in the Provider_ID column (for lack of a better way to identify the new provider who should own the service).
- Staff imports can modify any record, not just those owned by the importing user.

CHAPTER 5

Lifecycle

Lifecycle of the Service records

Service record statuses:

- current - visible to the public, provider, and staff after approval by IRC staff
- draft - pending review by IRC staff. visible to provider and staff.
- canceled - provider canceled review before IRC staff acted on it. visible to provider and staff. OR provider withdrew the service after it was approved.
- rejected - IRC staff rejected it visible to provider and staff.
- archived - no longer visible in most interfaces, just keeping around because there might be links back to it from JIRA etc.

Provider creates a new service. It needs approval:

pk: 1 status: draft

Three possible new states:

Provider cancels it:

pk: 1 status: canceled

Or, IRC rejects it:

pk: 1 status: reject

Or, IRC approves it:

pk: 1 status: current

From state current, provider submits some proposed changes. Now we have:

pk: 1 status: current

pk: 2 status: draft update_of: 1

Both of these records are visible to the provider, so they can see that they have a pending update.

From draft, pk 2 can go to canceled or rejected as before, which would leave pk 1 unchanged, but if it is approved then we don't need pk 1 anymore, so we archive it and nobody sees it anymore. We get:

pk: 1 status: archived

pk: 2 status: current update_of: 1

So now the only records anyone sees are:

pk: 2 status: current

Suppose provider had submitted a change but it's been rejected, so we have these two records lying around:

pk: 1 status: current

pk: 2 status: rejected update_of: 1

Now they submit a new proposed change. We only want 2 non-archived records around at any one time, so change the rejected record to archived before adding the new draft record:

pk: 1 status: current

pk: 2 status: archive update_of: 1

pk: 3 status: draft update_of: 1

The archive records don't show up anywhere for now, unless you have a link to them.

If we had a canceled record lying around, we'd do the same thing with it (change status to archived before proceeding).

Suppose the provider doesn't want their service published anymore. If we have

pk: 1 status: current

they can withdraw or cancel it, giving

pk: 1 status: canceled

leaving the service not visible to the public.

If the provider has a rejected or canceled record, they might reasonably want to edit it a little and "re-open" it, putting it back in draft status and requesting a new review.

But for phase I, let's not implement that to save time. We can always add it later.

The working spec says that even while a change is pending, the provider can edit the draft service record and an update will be sent to the JIRA ticket to let the staff know that the data has been updated again.

Here's that scenario with a new service pending. We start with:

pk: 1 status: draft update_of: none

Now the provider "edits" that draft, and we end up with:

pk: 1 status: archived

pk: 2 status: draft update_of: none

And we send a new comment to the JIRA ticket that was created when the first service record was submitted. The comment includes a link to the new record.

Now suppose we have one current record, and another record with a draft change to it:

pk: 1 status: current

pk: 2 status: draft update_of: 1

Now someone submits an edit to pk 2. We archive 2 and create 3:

pk: 1 status: current

pk: 2 status: archived update_of: don't care

pk: 3 status: draft update_of: 1

and we submit a comment to the JIRA issue from pk: 2 and include a link to pk: 3.

CHAPTER 6

Translation

Backend

Backend translation uses the standard Django translation mechanisms (https://docs.djangoproject.com/en/1.7/topics/i18n/).

Then we use Transifex to make it easy for client staff to provide translations of the text in the Django project.

Frontend

Frontend translation uses i18next-client. The source messages are in `frontend/locales/en/translation.json`.

Our fab helper commands convert those to **.po* files and push them to Transifex for translation, then convert the translated .po files to json for i18next-client to use.

Adding features and fixing bugs

While the `.po` files can be regenerated easily by running `fab makemessages` again for English or `fab pullmessages` for the translated languages, we still store them in Git to make it easier to keep an eye on changes, and maybe revert if we have to. (That way we are less likely to accidentally make a mistake and delete huge swaths of messages and not even notice it.)

We also store the `.mo` files because those are what Django gets the translated messages from at runtime.

Updating messages on Transifex

Anytime there have been changes to the messages in the code or templates that have been merged to develop, someone should update the messages on Transifex as follows:

1. Make sure you have the latest code from develop:

```
git checkout develop
git pull
```

2. regenerate the English (only) .po files:

```
fab makemessages
```

3. Run "git diff" and make sure the changes look reasonable.
4. If so, commit the updated .po file to develop and push it upstream:

```
git commit -m "Updated messages" locale/en/LC_MESSAGES/*.po
git push
```

   (Commiting the .po files isn't strictly necessary since we can recreate it, but we can tell what's the latest version we pushed to Transifex by committing each version when we push it.)

5. push the updated source file to Transifex (http://support.transifex.com/customer/portal/articles/996211-pushing-new-translations):

```
fab pushmessages
```

Updating translations from Transifex

Anytime translations on Transifex have been updated, someone should update our translation files on the develop branch as follows:

1. Make sure you have the latest code from develop:

```
git checkout develop
git pull
```

2. pull the updated .po files from Transifex (http://support.transifex.com/customer/portal/articles/996157-getting-translations):

```
fab pullmessages
```

3. Use `git diff` to see if any translations have actually changed. If not, you can stop here.
4. Also look at the diffs to see if the changes look reasonable. E.g. if a whole lot of translations have vanished, figure out why before proceeding.
5. Compile the messages to .mo files:

```
fab compilemessages
```

   If you get any errors due to badly formatted translations, open issues on Transifex and work with the translators to get them fixed, then start this process over.

6. Run your test suite one more time:

```
python manage.py test
```

7. Commit and push the changes to github:

```
git commit -m "Updated translations" locale/*/LC_MESSAGES/*.po locale/*/LC_
↪MESSAGES/*.mo
git push
```

CHAPTER 7

Server Provisioning

Overview

This project is deployed on the following stack.

- OS: Ubuntu 14.04 LTS
- Python: 3
- Database: Postgres 9.4
- Application Server: Gunicorn
- Frontend Server: Nginx
- Cache: Memcached

These services can be configured to run together on a single machine or on different machines. Supervisord manages the application server process.

We've used this AWS AMI:

```
ubuntu-trusty-14.04-amd64-server-20140927 (ami-b83c0aa5)
```

Initial Setup

This project uses Python 3. Because Fabric does not support Python 3, you will need Fabric installed on your laptop "globally" so that when you run `fab`, it will not be found in your virtualenv, but will then be found in your global environment:

```
sudo pip install fabric
```

For the environment you want to setup, you will need to set the `domain` in `conf/pillar/<environment>/env.sls`.

You will also need add the developer's user names and SSH keys to `conf/pillar/devs.sls`. Each user record (under the parent `users:` key) should match the format:

```
example-user:
  public_key:
    - ssh-rsa <Full SSH Public Key would go here>
```

Additional developers can be added later, but you will need to create at least one user for yourself.

Managing Secrets

Secret information such as passwords and API keys should never be committed to the source repository. Instead, each environment manages its secrets in `conf/pillar/<environment>/secrets.sls`. These `secrets.sls` files are excluded from the source control and need to be passed to the developers out of band. There are example files given in `conf/pillar/<environment>/secrets.ex`. They have the format:

```
secrets:
  DB_PASSWORD: XXXXXX
```

Each key/value pair given in the `secrets` dictionary will be added to the OS environment and can retrieved in the Python code via:

```
import os

password = os.environ['DB_PASSWORD']
```

Secrets for other environments will not be available. That is, the staging server will not have access to the production secrets. As such there is no need to namespace the secrets by their environment.

Environment Variables

Other environment variables which need to be configured but aren't secret can be added to the `env` dictionary in `conf/pillar/<environment>/env.sls`:

```
# Additional public environment variables to set for the project
env:
  FOO: BAR
```

For instance the default layout expects the cache server to listen at `127.0.0.1:11211` but if there is a dedicated cache server this can be changed via `CACHE_HOST`. Similarly the `DB_HOST/DB_PORT` defaults to `''/''`:

```
env:
  DB_HOST: 10.10.20.2
  CACHE_HOST: 10.10.20.1:11211
```

Setup Checklist

To summarize the steps above, you can use the following checklist

- `repo` is set in `conf/pillar/<environment>/env.sls`
- Developer user names and SSH keys have been added to `conf/pillar/devs.sls`

- Project name has been set in `conf/pillar/project.sls`
- Environment domain name has been set in `conf/pillar/<environment>/env.sls`
- Environment secrets including the deploy key have been set in `conf/pillar/<environment>/secrets.sls`

Salt Master

Each project needs to have at least one Salt Master. There can be one per environment or a single Master which manages both staging and production. The master is configured with Fabric. You will need to be able to connect to the server as a root user. How this is done will depend on where the server is hosted. VPS providers such as Linode will give you a username/password combination. Amazon's EC2 uses a private key. These credentials will be passed as command line arguments.

Template of the command:

```
fab -H <fresh-server-ip> -u <root-user> setup_master
```

Example of provisioning 33.33.33.10 as the Salt Master:

```
fab -H 33.33.33.10 -u root setup_master
```

Example AWS setup:

```
fab -H 54.235.72.124 -u ubuntu -i ~/.ssh/caktus-deployment.pem setup_master
```

This will install salt-master and update the master configuration file. The master will use a set of base states from https://github.com/caktus/margarita using the gitfs root. Once the master has been provisioned you should set:

```
env.master = '<ip-of-master>'
```

in the top of the fabfile.

If each environment has its own master then it should be set with the environment setup function `staging` or `production`. In these case most commands will need to be preceded with the environment to ensure that `env.master` is set.

Additional states and pillar information are contained in this repo and must be rsync'd to the master via:

```
fab -u <root-user> sync
```

This must be done each time a state or pillar is updated. This will be called on each deploy to ensure they are always up to date.

To provision the master server itself with salt you need to create a minion on the master:

```
fab -H <ip-of-new-master> -u <root-user> --set environment=master setup_minion:salt-
↪master
fab -u <root-user> accept_key:<server-name>
fab -u <root-user> --set environment=master deploy
# Example AWS setup
fab -H 54.235.72.124 -u ubuntu -i ~/.ssh/caktus-deployment.pem --set␣
↪environment=master setup_minion:salt-master
fab -H 54.235.72.124 -u ubuntu -i ~/.ssh/caktus-deployment.pem --set␣
↪environment=master deploy
```

This will create developer users on the master server so you will no longer have to connect as the root user.

Provision a Minion

Once you have completed the above steps, you are ready to provision a new server for a given environment. Again you will need to be able to connect to the server as a root user. This is to install the Salt Minion which will connect to the Master to complete the provisioning. To setup a minion you call the Fabric command:

```
fab <environment> setup_minion:<roles> -H <ip-of-new-server> -u <root-user>
fab staging setup_minion:web,balancer,cache -H  33.33.33.10 -u root
# Example AWS setup
fab staging setup_minion:web,balancer,cache,queue,worker -H 54.235.72.124
```

The available roles are `salt-master`, `web`, `worker`, `balancer`, `queue` and `cache`. If you are running everything on a single server you need to enable the `web`, `balancer`, and `cache` roles. The `worker` and `queue` roles are only needed to run Celery which is explained in more detail later.

The IRC deploy uses a database on another server, so a `db-master` role is not needed.

Additional roles can be added later to a server via `add_role`. Note that there is no corresponding `delete_role` command because deleting a role does not disable the services or remove the configuration files of the deleted role:

```
fab add_role:web -H  33.33.33.10
```

After that you can run the deploy/highstate to provision the new server:

```
fab <environment> deploy
```

The first time you run this command, it may complete before the server is set up. It is most likely still completing in the background. If the server does not become accessible or if you encounter errors during the process, review the Salt logs for any hints in `/var/log/salt` on the minion and/or master. For more information about deployment, see the *server setup </server-setup>* documentation.

Optional Configuration

The default template contains setup to help manage common configuration needs which are not enabled by default.

HTTP Auth

The `secrets.sls` can also contain a section to enable HTTP basic authentication. This is useful for staging environments where you want to limit who can see the site before it is ready. This will also prevent bots from crawling and indexing the pages. To enable basic auth simply add a section called `http_auth` in the relevant `conf/pillar/<environment>/secrets.sls`:

```
http_auth:
  admin: 123456
```

This should be a list of key/value pairs. The keys will serve as the usernames and the values will be the password. As with all password usage please pick a strong password.

Celery

Many Django projects make use of Celery for handling long running task outside of request/response cycle. Enabling a worker makes use of Django setup for Celery. As documented you should create/import your Celery app in `service_info/__init__.py` so that you can run the worker via:

```
celery - A worker
```

Additionally you will need to configure the project settings for Celery:

```
# service_info.settings.staging.py
import os
from service_info.settings.base import *

# Other settings would be here
BROKER_URL = 'amqp://service_info_staging:%(BROKER_PASSWORD)s@%(BROKER_HOST)s/service_
↪info_staging' % os.environ
```

You will also need to add the `BROKER_URL` to the `service_mapper.settings.production` so that the vhost is set correctly. These are the minimal settings to make Celery work. Refer to the Celery documentation for additional configuration options.

`BROKER_HOST` defaults to `127.0.0.1:5672`. If the queue server is configured on a separate host that will need to be reflected in the `BROKER_URL` setting. This is done by setting the `BROKER_HOST` environment variable in the `env` dictionary of `conf/pillar/<environment>/env.sls`.

To add the states you should add the `worker` role when provisioning the minion. At least one server in the stack should be provisioned with the `queue` role as well. This will use RabbitMQ as the broker by default. The RabbitMQ user will be named service_info_<environment> and the vhost will be named service_info_<environment> for each environment. It requires that you add a password for the RabbitMQ user to each of the `conf/pillar/<environment>/secrets.sls`:

```
secrets:
  BROKER_PASSWORD: thisisapasswordforrabbitmq
```

The worker will run also run the `beat` process which allows for running periodic tasks.

SSL

The default configuration expects the site to run under HTTPS everywhere. However, unless an SSL certificate is provided, the site will use a self-signed certificate. To include a certificate signed by a CA you must update the `ssl_key` and `ssl_cert` pillars in the environment secrets. The `ssl_cert` should contain the intermediate certificates provided by the CA. It is recommended that this pillar is only pushed to servers using the `balancer` role. See the `secrets.ex` file for an example.

You can use the below OpenSSL commands to generate the key and signing request:

```
# Generate a new 2048 bit RSA key
openssl genrsa -out service_info.key 2048
# Make copy of the key with the passphrase
cp service_info.key service_info.key.secure
# Remove any passphrase
openssl rsa -in service_info.secure -out service_info.key
# Generate signing request
openssl req -new -key service_info.key -out service_info.csr
```

The last command will prompt you for information for the signing request including the organization for which the request is being made, the location (country, city, state), email, etc. The most important field in this request is the common name which must match the domain for which the certificate is going to be deployed (i.e example.com).

This signing request (.csr) will be handed off to a trusted Certificate Authority (CA) such as StartSSL, NameCheap, GoDaddy, etc. to purchase the signed certificate. The contents of the `*.key` file will be added to the `ssl_key` pillar and the signed certificate from the CA will be added to the `ssl_cert` pillar.

Copying production database and media to staging

The server should be stopped before performing this procedure and restarted afterwards. The procedure uses commands in the ServiceInfo-ircdeploy repository:

```
$ cd ServiceInfo-ircdeploy
$ workon virtualenv-with-fab
$ fab staging refresh_environment
$ fab staging "manage_run:change_cms_site --from=serviceinfo.rescue.org --
↪to=serviceinfo-staging.rescue.org"
$ fab staging "manage_run:rebuild_index --noinput"
```

CHAPTER 8

Server Setup

Provisioning

The server provisioning is managed using Salt Stack. The base states are managed in a common repo and additional states specific to this project are contained within the `conf` directory at the root of the ServiceInfo-ircdeploy repository.

For more information see the doc:*provisioning guide </provisioning>*.

Layout

Below is the server layout created by this provisioning process:

```
/var/www/service_info/
    source/
    env/
    log/
    public/
        static/
        media/
    ssl/
```

`source` contains the source code of the project. `env` is the virtualenv for Python requirements. `log` stores the Nginx, Gunicorn and other logs used by the project. `public` holds the static resources (css/js) for the project and the uploaded user media. `public/static/` and `public/media/` map to the `STATIC_ROOT` and `MEDIA_ROOT` settings. `ssl` contains the SSL key and certificate pair.

Deployment

For deployment, each developer connects to the Salt master as their own user. Each developer has SSH access via their public key. These users are created/managed by the Salt provisioning. The deployment itself is automated with Fabric. To deploy, a developer simply runs:

```
# Deploy updates to staging
fab staging deploy
# Deploy updates to production
fab production deploy
```

This runs the Salt highstate for the given environment. This handles both the configuration of the server as well as updating the latest source code. This can take a few minutes and does not produce any output while it is running. Once it has finished the output should be checked for errors.

New server on AWS

1. Create a new EC2 server. Some tips:

- Put it in a region close to where most users will be, e.g. Ireland (eu-west-1). (To switch regions in the AWS EC2 console, look near the top-right of the window for a light-gray selector on a black background.)
- Use an AMI (image) of Ubuntu 14.04 server, 64-bit, EBS - e.g. ubuntu-trusty-14.04-amd64-server-20140927 (ami-b83c0aa5)
- Be sure to save the private key that is created, or use an existing one you already own. (Caktus: key pairs are stored in LastPass, search for CTS.) The AWS private key is only needed until CTS has been deployed the first time, but it is essential until then.

1. If needed, add a new environment in the fabfile and Salt config files.
2. Add the new server's ssh key to your ssh-agent, e.g.:

   ```
   ssh-add /path/to/newserver.pem
   ```

   This will allow you to ssh into the new server as `ubuntu` initially. After we've finished our deploy, you'll have your own userid on the server that you can use to ssh in.
3. Create a master:

   ```
   fab -u ubuntu staging setup_master
   ```

4. Create a minion and assign initial roles:

   ```
   fab -u ubuntu staging setup_minion:balancer,queue,cache,web,worker,beat
   ```

5. Initial deploy:

   ```
   fab -u ubuntu staging deploy
   ```

After that, developer accounts will exist on the server with ssh access, so "-u ubuntu" will no longer be needed. You'll be able to update the server with:

```
fab staging deploy
```

CHAPTER 9

Vagrant Testing

Starting the VM

You can test the provisioning/deployment using Vagrant. This requires Vagrant 1.3+. The Vagrantfile is configured to install the Salt Master and Minion inside the VM once you've run `vagrant up`. The box will be installed if you don't have it already.:

```
vagrant up
```

The general provision workflow is the same as in the previous *provisioning guide* so here are notes of the Vagrant specifics.

Provisioning the VM

Set your environment variables and secrets in `conf/pillar/vagrant.sls`. It is OK for this to be checked into version control because it can only be used on the developer's local machine. To finalize the provisioning you simply need to run:

```
fab vagrant_first_time deploy
```

and thereafter:

```
fab vagrant deploy
```

The Vagrant box will use the current working copy of the project and the vagrant.py settings. If you want to use this for development/testing it is helpful to change your local settings to extend from staging instead of dev:

```
# Example local.py
from service_info.settings.staging import *

# Override settings here
DATABASES['default']['NAME'] = 'service_info_local'
```

```
DATABASES['default']['USER'] = 'service_info_local'

DEBUG = True
```

This won't have the same nice features of the development server such as auto-reloading but it will run with a stack which is much closer to the production environment. Also beware that while `conf/pillar/vagrant.sls` is checked into version control, `local.py` generally isn't, so it will be up to you to keep them in sync.

Testing on the VM

With the VM fully provisioned and deployed, you can access the VM at the IP address specified in the `Vagrantfile`, which is 33.33.33.10 by default. Since the Nginx configuration will only listen for the domain name in `conf/pillar/vagrant.sls`, you will need to modify your `/etc/hosts` configuration to view it at one of those IP addresses. I recommend 33.33.33.10, otherwise the ports in the localhost URL cause the CSRF middleware to complain `REASON_BAD_REFERER` when testing over SSL. You will need to add:

```
33.33.33.10 <domain>
```

where `<domain>` matches the domain in `conf/pillar/vagrant.sls`. For example, let's use dev.example.com:

```
33.33.33.10 dev.example.com
```

In your browser you can now view https://dev.example.com and see the VM running the full web stack.

Note that this `/etc/hosts` entry will prevent you from accessing the true dev.example.com. When your testing is complete, you should remove or comment out this entry.

CHAPTER 10

ServiceInfo backups

Getting a backup dump and restoring it locally

Steps you can do ahead of time:

- Get access to the Caktus backup server (open a tech support request).

When you need to restore a backup:

- Make sure you are in your ServiceInfo project directory, and not in the ServiceInfo-IRCDeploy project directory:

```
$ git config --get remote.origin.url
git@github.com:theirc/serviceinfo.git
```

- List the files in the latest backup directory and find the most recent:

```
$ backup_path=/mnt/rsnapshot/serviceinfo-prod/hourly.0/home/caktus-backup
$ backup_filename=`ssh caktus-backup@backup.caktus.lan ls $backup_path | tail -1`
```

- Copy that file to your local directory and MAKE SURE it's named `service_info.sql.bz2`. That's not important if you're only doing a local restore, but if you're restoring to staging, our Fabric scripts look for that particular filename.:

```
$ scp caktus-backup@backup.caktus.lan:${backup_path}/${backup_filename} service_
↪info.sql.bz2
```

It's about 3.7 MB as of Sept 2017.

- Decompress the file:

```
$ bunzip2 service_info.sql.bz2
```

Now the file is in `service_info.sql` (in SQL text format).

Drop your existing local database and restore from the backup:

```
$ dropdb service_info
$ createdb --template=template0 service_info
$ psql service_info < service_info.sql
```

There will be a bunch of errors related to the fact that the production dump has roles which your local DB doesn't have. It's OK to ignore them.

Migrate the database:

```
$ workon serviceinfo
$ ./manage.py migrate --noinput
```

Next, we need to update the site name in the CMS data. The value 'localhost:8000' is special because the 'Domain Name' is hard-coded to that value in the Django admin ("Sites" app):

```
$ ./manage.py change_cms_site --from=serviceinfo.rescue.org --to=localhost:8000
```

Update your media directory with the media from production:

```
$ rsync -zPa -e ssh --delete caktus-backup@backup.caktus.lan:/mnt/rsnapshot/
↪serviceinfo-prod/hourly.0/var/www/service_info/public/media public
```

Now run the server normally and visit 'http://localhost:4005/' to make sure things look OK:

```
$ gulp
```

Continue to the next section to restore the staging server. When you're done, be sure to delete the dump from your laptop:

```
$ rm -f service_info.sql
```

Bringing up a new site using the backup dump

The script to update the staging server expects that you have completed the steps above to copy the database and media backups to the ServiceInfo project directory.

Now switch over to the ServiceInfo-IRCDeploy project:

```
$ cd ../ServiceInfo-ircdeploy/
$ git config --get remote.origin.url
git@github.com:theirc/ServiceInfo-ircdeploy.git
```

Run the `refresh_from_backup` command. It takes one parameter, which is an absolute or relative path to the main ServiceInfo project directory on your laptop (Mine is in a sibling directory named `serviceinfo`):

```
$ workon virtualenv-with-fab
$ fab staging refresh_from_backup:../serviceinfo
```

This command uploads the database dump and media to staging, stops the web server, imports the dump, runs migrations, resets the CMS, rebuilds the ElasticSearch index, and then restarts the web server.

There is a LOT of output from this command, including the rsync progress, and multiple warnings and errors from the DB import. There is also a warning about missing migrations which are due to a Python 2/3 incompatibility in some third-party packages. That can be ignored for now.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search