LexPredict ContraxSuite Quick Deployment

Development Guide
Release 1.0.8 - April 1, 2018

How to create and plugin a new application.

1. Create a new application in `contraxsuite_services/apps/` directory.
2. Use files structure similar to other applications (f.e. "documents").
3. Add custom app into `INSTALLED_APPS` in `settings.py`.
4. Create your templates in `apps/your_app/templates` folder.
5. To insert a menu item in main menu for custom app, create template `apps/your_app/templates/main_menu_item.html` and use there `<li>` tag structure similar to existing in main menu (see `templates/_base_menu_style1.html`).
In that case it will be displayed in main menu,
as first item. If a project has more than one custom app,
e.g. "employment" + "leases", then there will be displayed "Contract Types"
menu item and it will have subitems, defined in
`apps/employment/templates/main_menu_item.html` and
`apps/leases/templates/main_menu_item.html` templates.
6. Create custom admin tasks in `apps/your_app/tasks.py`, inherit those tasks
from `apps.task.tasks.BaseTask`. Register custom class-based task.
Use unique names for admin tasks. See examples in `apps.task.tasks`.
7. To insert a menu item in task menu on Task List page for custom app,
create template `apps/your_app/templates/task_menu_item.html`
and use there `<li>` tag structure similar to existing in task menu.
(see `templates/task/task_list.html`).
In that case it will be displayed in task menu, as second item.
If a project has more than one custom app, their menu items will be displayed
in task menu starting from second item.
8. Any template from main `templates` folder can be overwritten,
just create template with the same name and place it under your templates directory:
e.g. `apps/your_app/templates/document/stats.html`.
NOTE: your app should be latest in `INSTALLED_APPS` in `settings.py`.
9. There is no need to include urls from custom app in main `urls.py` -
they will be added automatically using namespace equal to an app name.
E.g. for `an_app` custom application under `apps` directory,
(if it's included in `INSTALLED_APPS` and it has it's own `urls.py`) -
there will be automatically included urls for that app in main `urls.py`
like `/an_app/...`.

How to customize locators.

1. This project has some buitin locators.
2. Specify *"required"* locators in `REQUIRED_LOCATORS` in `setting.py`.
Users won't be able to disable these locators using "App configuration" menu.
3. Specify *"optional"* locators in `OPTIONAL_LOCATORS` setting.
These locators can be disabled. In this case they won't be available
in Task menu, items related to these locators won't be displayes in main menu > Extract menu.
4. Specify `LOCATOR_GROUPS` in `settings.py` and disable locators in
`Extract` menu using group name (see `templates/_base_menu_style1.html`).
5. See `apps.common.middleware.AppEnabledRequiredMiddleware`.

Autologin Middleware.

- You can enable "auto login" via web "App configuration" menu
  (see "Auto Login" config menu setting).
- In this case if you navigate to any page except those declared in
  `AUTOLOGIN_ALWAYS_OPEN_URLS` (for now it is only login page)
  you will be signed in as "test_user".
- "test_user" is a user with role "manager" for demo/test purposes.
- You can forbid any url in `AUTOLOGIN_TEST_USER_FORBIDDEN_URLS` for test user.
  (see `apps.common.middleware.AutoLoginMiddleware` for details).

App configuration menu.

1. Admin users have access to "App configuration" menu (see "gears" icon in top right corner).
2. The App uses `django-constance` for it:
   see https://django-constance.readthedocs.io/en/latest/
3. There are some settings which can be configured on the fly:
   - enable/disable "autologin" feature (see above);
   - enable/disable standard optional locators and related main menu items.

Minifying traffic (html/js/css).

1. Minifying html:
   - enabled using `django.middleware.gzip.GZipMiddleware` in `MIDDLEWARE` setting
     in `settings.py`. It compresses html content using gzip compressor.
   - This reduces html file content length by 75-80%. Transferred file will have `Content-Encoding:gzip`.
   - See https://docs.djangoproject.com/en/2.0/ref/middleware/#module-django.middleware.gzip.
2. Minifying css/js:
   - enabled using `django-pipeline`;
   - `PIPELINE_ENABLED = True` enables css/html minifying,
     just overwrite it in `local_settings` if needed.

- `PIPELINE` dictionary defines which files should be minified.
- After all files added to `PIPELINE` you must run `collectstatic` procedure to produce minified files.
- To test `django-pipeline` locally copy minified `pipeline` files from internal `staticfiles` directory into `static` folder according to paths specified in `PIPELINE`, because django serves files from initial `static` folder during development.
- This reduces content length by 5-10%.
- WARNING! Be careful and test any added to `PIPELINE` file before deploying on production.
- See https://django-pipeline.readthedocs.io/en/latest/ for details.

Authentication.

1. The App uses `django-allauth` for authentication,
see http://django-allauth.readthedocs.io/en/latest/overview.html.
2. See settings which start with `ACCOUNT_`, see
https://django-allauth.readthedocs.io/en/latest/configuration.html.
3. The App uses `apps.common.middleware.LoginRequiredMiddleware`, it
requires a user to be authenticated to view any page other than LOGIN_URL.
4. To create a superuser use this management command:
`./manage.py create_superuser --username USER --password PWD --email it@email.com`
5. To create a user in python shell:
    - create a user (with role) + set password;
    - create EmailAddress for it where `verified=True`
6. Basically the App has some predefined user roles:
    - reviewer (without admin access rights);
    - manager (with manager admin rights, but without superuser access);
    - technical_admin (admin + superuser access rights)
    see `apps.user.models.User`

Permissions.

1. The App uses permissions based on user's role (see #6 above).
2. The App has some permission mixins defined in `apps/common/mixins.py`,
see `AdminRequiredMixin`, `TechAdminRequiredMixin`, `ReviewerQSMixin`.
3. Basically access to documents / text units / extracted items depends on
user role:
    - `the_admin` and `manager` have access to all documents;
    - `reviewer` has access to those documents which assigned to a task queue
    where that reviewer is present in `TaskQueue.reviewers`
    - see `apps.common.mixins.ReviewerQSMixin`, use `limit_reviewers_qs_by_field` to
    filter a queryset.
    - use `is_admin`, `is_reviewer`, `is_manager` methods from `apps.users.models.User` to
    allow/disallow access for certain roles
    - use `User.can_view_document(Document)` from `apps.users.models.User` to
    define/check user permissions for concrete document

Two-Factor Authentication (TFA).

1. To enable TFA:
    - set `ACCOUNT_ADAPTER = 'allauth_2fa.adapter.OTPAdapter'` in settings/local_settings (it is redefined in local_setting as `apps.users.adapters.AccountAdapter` for now);
    - install "Google Authenticator" on your cell phone;
    - go to profile page and click "TWO FACTOR AUTHENTICATION" button;
    - scan the QR code on the page with a token generator installed on your cell phone;
    - input a token generated by the "Google Authenticator" on "TFA" page;
    - you will be prompted to generate backup tokens;
    - generate and store backup tokens;
2. Now each time you want to login you need to enter a token generated by "Google Authenticator" on your cell phone;
3. WARNING! There may be issues related with date/time synchronization between contraxsuite server and your device, use synchronization menu in "Google Authenticator" app.
4. See https://pypi.python.org/pypi/django-allauth-2fa/ for details.

Asynchronous admin tasks.

1. The App uses `celery` for asynchronous tasks (see http://www.celeryproject.org/), celery app is declared in `apps/celery.py`
2. The App uses `RabbitMQ` as message broker for celery, see `CELERY_BROKER_URL` setting.
   To install `RabbitMQ` see https://github.com/LexPredict/lexpredict-contraxsuite-deploy/blob/master/fabfile.py#L850
   (`rabbitmq_install` method)
3. The App uses django db as result backend for celery, see `CELERY_RESULT_BACKEND` setting,
   see http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html#django-celery-results-using-the-django-orm-cache-as-a-result-backend
   Task results are available as `TaskResult` objects:
   `from django_celery_results.models import TaskResult`.
4. The App uses custom model for user tasks - see `apps.task.models.Task`.
5. Each task should be defined in `tasks.py` file and registered:
   use `app.register_task(ClassBasedTask())` or `app.tasks.register(ClassBasedTask())` for class-based tasks and `@shared_task` decorator for functions.
6. Class-based tasks should be inherited from `apps.task.tasks.BaseTask`, in this case a user can monitor their execution on Task List page.
    - put your staff into `process` method;
    - use `self.log(message)` for logging;
    - `self.task` represents `apps.task.models.Task` instance
    - use `self.task.subtasks_total` to set total amount of subtasks
    - save `Task` using `self.task.save()`
    - use `self.task.push()` to mark a *subtask* completed
    - use `self.task.force_complete()` to mark `Task` completed
    - `Task` can have status `PENDING`, `FAILURE`, `SUCCESS`, for details see `apps.task.models.Task`

    - `SUCCESS` status means that all subtasks are completed, in this case `Task.progress` is equal 100%

7. Example:

```
from django_celery_results.models import TaskResult
from celery.result import AsyncResult
from apps.task.models import Task
task = Task.objects.order_by('pk').last()
print(task.status)
task_result = task.celery_task
print(task_result.result)
async_result = AsyncResult(task_result.id)
child_tasks = async_result.children
for child_task in child_tasks:
    child_task.revoke(terminate=True)
```

8. Useful celery commands:
    - start celery app: `celery multi start 2 -A apps -f celery.log -B`,
     where `2` is number of workers
    - stop celery app: `celery multi stop 2 -A apps`
    - show status: `celery status -A apps`
    - show active tasks: `celery inspect active -A apps`
    - show registered tasks `celery inspect registered -A apps`
    - purge all tasks: `celery purge -A apps -f`
9. Use `sudo rabbitmqctl command` for RabbitMQ where `command` is your custom command,
  f.e. `status`. See https://www.rabbitmq.com/rabbitmqctl.8.html.

Files management.

1. The App uses `Django-filebrowser` to serve file uploading.
See https://django-filebrowser.readthedocs.io/en/latest/.
We use this fork which removes the dependency on Grappelli:
https://github.com/smacker/django-filebrowser-no-grappelli.
2. Filebrowser views use admin template layout, but adapted for using
in the App. Only superusers have access to filebrowser views for now.
This dependency can be eliminated: see `CAEQ` branch.
3. Filebrowser settings:
    - `FILEBROWSER_DIRECTORY = 'data/documents/'` - all files will be uploaded in this folder;
    - see settings which start with `FILEBROWSER_`,
    https://django-filebrowser.readthedocs.io/en/latest/settings.html

JqWidgets tips and tricks.

1. The App uses `JQWidgets` JavaScript & HTML5 UI Framework for building
highly customizable tables and charts.
See https://www.jqwidgets.com/.

2. It requires commercial license for commercial websites, so we
cannot distribute it.
3. To install jqwidgets see
https://github.com/LexPredict/lexpredict-contraxsuite-deploy/blob/master/fabfile.py#L1113
(`jqwidgets_install` method)
4. JqWidgets settings:
    - `JQ_EXPORT = False` - use jqWidgets' export or not, e.g. send data to jq OR handle it on client side
    see http://www.jqwidgets.com/community/topic/jqxgrid-export-data/

Exporting data from JqWidgets tables.

1. The App uses `django-excel` to export data (from "jqwidgets" tables)
if native JqWidgets export isn't available, see `JQ_EXPORT = False`,
2. See `FILE_UPLOAD_HANDLERS` setting.
3. See `apps.common.utils.export_qs_to_file` and use of it in `apps.common.mixins`
4. See http://django-excel.readthedocs.io/en/latest/.

Loading dictionary data for Courts, Terms, etc.

1. The App uses this repo: https://github.com/LexPredict/lexpredict-legal-dictionary
2. `GIT_DATA_REPO_ROOT` setting specifies that path and branch name.
3. But a user can *additionally* specify local file path/name to upload data from there as well;
file will be searched in these directories:
    - `<DATA_ROOT>`
    - `<MEDIA_ROOT>/<FILEBROWSER_DIRECTORY>`
4. See `apps.task.tasks.LoadTerms` task for example.

Elasticsearch (ES) integration.

1. The App uses ES in "Global Search" section on Text Unit List page.
2. See how to install ES:
https://github.com/LexPredict/lexpredict-contraxsuite-deploy/blob/master/fabfile.py#L890
(`elasticsearch_install` method)
3. The App uses `elasticsearch` python package - client for ES,
see https://elasticsearch-py.readthedocs.io/en/master/
4. See `ELASTICSEARCH_CONFIG` setting for configuring ES.
5. Use `sudo systemctl command elasticsearch` where command may be
`status`/`start`/`stop`/`restart`/etc

OCR Documents (using Tika & Textract).

1. The App uses `tika` and `textract` to ocr files.
See https://github.com/chrismattmann/tika-python,
and https://textract.readthedocs.io/en/stable/.
2. Tika works x5 times faster than textract and it's used by default.

3. Python-tika uses tika-server, there is another option - `tika-app`,
 but it works slowly ~ x4 times.
4. Python-tika extracts `tika-server.jar` if needed and run tika server,
 so parsing a document first time can take a little bit more time.
5. Python-tika requires `/tmp/tika.log` file, so check that it exists and available for
 a current user (usually ubuntu:ubuntu).
6. Textract installation tips / utils: see `apps.task.utils.ocr.textract`.

Django Admin site.

1. The App uses `django-suit` - modern theme for Django admin interface.
2. We can override admin site templates in `templates/admin/` folder
3. See http://djangosuit.com/,
 https://docs.djangoproject.com/en/2.0/ref/contrib/admin/

Django initial migrations.

1. Relationships between database tables are very complex,
and initial migrations don't pass django's system checks when migrations
are ran first time.
2. For this case we have custom management command which runs migrations
without system checks: `./manage.py force_migrate`
3. See `apps/common/management/commands/force_migrate.py` for details.
4. We run this command while initial deployment: see `manage('force_migrate')` in fabfile.

Historical objects.

1. The App uses `django-simple-history` to track history for DocumentNote objects.
2. See `apps.document.models.DocumentNote`, - each note has own `history` object
which represents a history of creating and changing that note.
3. See https://django-simple-history.readthedocs.io/en/latest/

Deployment.

1. Deploying via docker image:
The app now uses continuous integration of deployment process via jenkins
 and docker image.
 Read more about deploying using docker here:
 https://github.com/LexPredict/lexpredict-contraxsuite/blob/master/docker/QUICK_DEPLOY.md

2. Deploying via Fabric:
 * You can use `Fabric` for deployment, see http://www.fabfile.org/.
 * But `Fabric` doesn't support python3, so we use this fork
 https://pypi.python.org/pypi/Fabric3/1.10.2 which is compatible with python3.
 * Working with fabric:
 - all commands are in `deploy/fabfile.py`
 - for each instance create a folder `deploy/YOUR_INSTANCE_NAME`

- put 3 files there:
    - \`fabrirc\` which is configuration file with set of variables
    - \`your.pem\` ssh key
    - \`local_setting.py\` where you can redefine django variables from \`settings.py\`
- from \`deploy\` directory run command \`fab -c YOUR_INSTANCE_NAME/fabricrc COMMAND_NAME\`
- f.e. \`fab -c dev-alpha/fabricrc restart\` or \`fab -c demo/fabricrc manage:migrate\`
- see \`fabric\` documentation and \`fabfile.py\` commands for more information

Django settings.py and local_settings.py files.

1. The App has \`settins.py\` with all django settings.
2. To redefine django settings we use \`local_settings.py\` file,
just write there you own settings and put it next to \`settings.py\`
3. F.e. we can set \`DEBUG = True\` in local setting while main setting
has \`DEBUG = False\`.
4. You can use \`LOCAL_INSTALLED_APPS\` and \`LOCAL_MIDDLEWARE\` in local settings,
they will be concatenated with \`INSTALLED_APPS\` and \`MIDDLEWARE\` variables from
main settings file.
5. Use local settings for all secure settings like keys, passwords, credentials, etc.
Set there email backend, define database creds.
6. Include required hosts in \`ALLOWED_HOSTS\`: for local development include
\`localhost\` and \`127.0.0.1\`, for deployment use ip / dns names.

Useful python packages for development.

1. \`django-debug-toolbar\` - https://django-debug-toolbar.readthedocs.io/en/stable/
    - allows track sql queries efficiency
    - gives access to templates and their context
    - info about request, headers, django settings
2. \`django-extensions\` - https://django-extensions.readthedocs.io/en/latest/ -
is a collection of custom extensions for Django.
    - \`./manage.py shell_plus\`, \`./manage.py runserver_plus\`, etc
3. \`ipdb\` - https://pypi.python.org/pypi/ipdb - ipython debugger.
4. \`flower\` - http://flower.readthedocs.io/en/latest/ - celery monitoring tool