colab Documentation

Release 2.0dev

Sergio Oliveira

Sep 27, 2017

Contents

Colab is an integration server meant to help develpers to unify the **User Experience** in Web applications.

Colab provides ways to integrate:

- Authentication or Single Sign-On (SSO)
- User Interface (UI)
- Data

To accomplish that Colab is placed in front of integrated Web applications. All user requests and responses are proxied (as in the image *Colab Reverse Proxy model*) and therefore can have content and headers modified. Also, due to it's architecture, Colab can grant or deny access to systems under it.

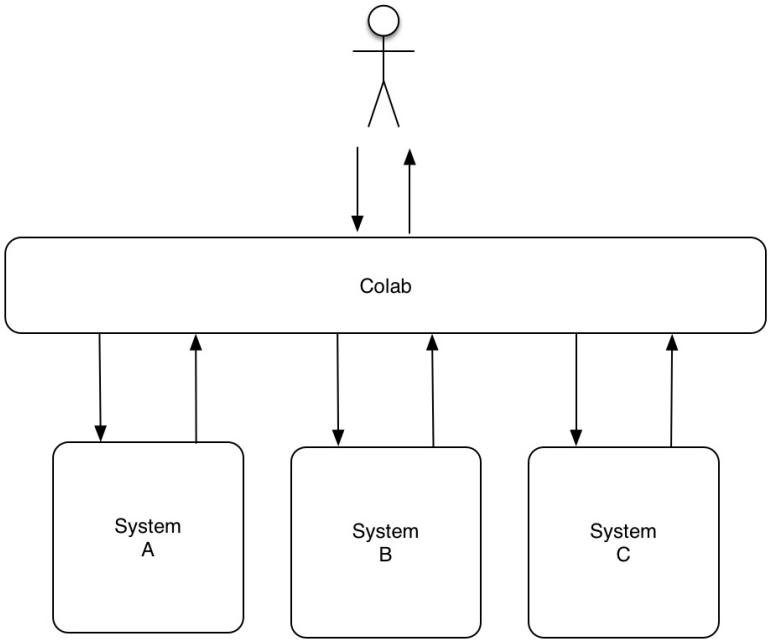

Fig. 1: *Colab Reverse Proxy model*

Plugins are used in order to integrate new Web applications. Currently the following plugins are available and maintained by core developers:

- Mailman
- Gitlab
- Noosfero
- Mezuro

If you need to integrate a diffirent tool please refer to *Plugin Developer Documentation*.

Contents:

CHAPTER 1

User Documentation

Getting Started

Install Requirements with Vagrant

Need VirtualBox and Vagrant installed

Run the following command to create and up the colab virtual machine

```
$ vagrant up
```

Run the following command to access colab virtual machine

```
$ vagrant ssh
```

Install Requirements without Vagrant

Install virtualenvwrapper

Use this link to configure the virtualenvwrapper: https://virtualenvwrapper.readthedocs.org

Run the following command

```
$ mkvirtualenv colab
```

Install Development

On the colab folder use the following commands, or in the colab vagrant ("$ vagrant ssh")

```
$ workon colab
$ pip install -e . #(dont need this command if use vagrant)
```

```
$ colab-admin migrate #(dont need this command if use vagrant)
$ colab-admin runserver 0.0.0.0:8000
```

Colab settings

View the following file:

```
$ cat /etc/colab/settings.py
```

The file /etc/colab/settings.py have the configurations of colab, this configurations overrides the django settings.py

Colab can generate its own default settings file for development environment with

```
$ colab-admin initconfig > /etc/colab/settings.py
```

Colab development settings - Log Level

If you used the default settings file from initconfig. You can especify the environment variable **COLAB_LOGLEVEL** to set which log level you want to show on the console.

You can also set **COLAB_DEBUG** that is an alias to *COLAB_LOGLEVEL=DEBUG*

The following log levels exist:

- **DEBUG**: Show everything possible.
- **INFO**: Show info, and errors.
- **ERROR**: Default behavior, show errors, warnings and critical messages.

Example of running runserver showing **INFO** logs:

```
$ COLAB_LOGLEVEL=INFO colab-admin runserver 0.0.0.0:8000
```

Widgets

A widget is a piece of HTML that will be inserted in a specific area in a page to render some view.

To configure the widgets you have to edit, or create, the file `/etc/colab/widgets_settings.py`. Or you can create a py file inside the folder `/etc/colab/widgets.d`.

Colab can generate its own default widgets settings file for development environment with

```
$ colab-admin initwidgetsconfig > /etc/colab/widgets_settings.py
```

Example:

```
# Widget Manager handles all widgets and must be imported to register them
from colab.widgets.widget_manager import WidgetManager

# Specific code for Gitlab's Widget
from colab_gitlab.widgets import GitlabProfileWidget

WidgetManager.register_widget('profile', GitlabProfileWidget())
```

In this example the Gitlab's widget is added in a new tab inside the user profile.

Widgets Areas

List
The list widget area can be found at the user profile. It provides general information like latest news and collaboration.

Button
The button widget area can be found at the user profile. It provides aditional buttons to profile.

Group
The group widget area can be found at the user profile. It provides aditional information to profile.

Dashboard
The dashboard widget area can be found at `/dashboard`. It provides general information like latest news and collaboration.

Charts
The charts widget area can be found at the user profile page. It provides an area for charts displays.

Core Widgets

Most Relevant Threads
Shows the list of most relevant threads.

```
# Widget Manager handles all widgets and must be imported to register them
from colab.widgets.widget_manager import WidgetManager

# Specific code for Gitlab's Widget
from colab_gitlab.widgets import GitlabProfileWidget

WidgetManager.register_widget('profile', GitlabProfileWidget())
```

Latest Threads
Shows the list of latest threads, those threads are get from the public mailling lists.

Groups
Shows the groups that user is subscribed.

Suggested area: group

Group Membership
Adds a button to subscribed in a group.

Suggested area: button

Latest Posted
Shows the list of latest post, those post are get from the public mailling lists from the user.

Suggested area: list

Your Latest Contributions
Shows the list of latest contributions for the user.

Suggested area: list

Latest Collaborations
Shows the list of latest collaborations, in example, articles and blog post done recently.

Suggested area: dashboard

Collaboration Graph
Displays a pie chart of all collaborations that are indexed.

Suggested area: dashboard

Collaboration Chart
Displays a pie chart of all collaborations created by user that are indexed.

Suggested area: charts

Participation Chart
Displays a pie chart of all participations from mail list.

Suggested area: charts

Add a new plugin

- Atention: replace the brackets, [], for the content presented in the brackets
- Make sure the application has the following requirements
  - Support for remote user authentication
  - A relative url root
  - A relative static url root, for change url's of css and javascript
- Create the plugin configuration for the application
  - on folder: /etc/colab/plugins.d/
  - create file: [plugin_name].py
- Atention: Any URL used in the plugins' settings should not be preceded by "/"

Use this template for the plugin configuration file

```
from colab.plugins.utils.menu import colab_url_factory
from django.utils.translation import ugettext_lazy as _

name = 'colab.plugins.[plugin_name]'

upstream = 'http://[host_of_application]/[relative_url_root]/'

# The private_token is optional
# It is used to access the application data being coupled to colab
# It is recommended to use the provate_token an admin of the application
private_token = '[plugin_private_token_for_data_import]'

urls = {
    'include': '[plugin_module_path].urls',
    'prefix': '[application_prefix]/', # Exemple: http://site.com/[application_
↪prefix]/
}

menu_title = '[menu_title_of_html]'

url = colab_url_factory('[plugin_name]')

menu_urls = {
    url(display=_('[name_of_link_page]'), viewname='[name_of_view_in_the_application]
↪', kwargs={'path': '[page_appication_path]/' }, auth=True),
```

```
    # You can have more than one url
    url(display=_('[name_of_link_page]'), viewname='[another_name_of_view_in_the_
↪application]', kwargs={'path': '[another_page_appication_path]/' }, auth=True),
}
```

Plugins

name

Declares the absolute name of the plugin app as a python import path. Example: directory.something.someplugin

verbose_name

Delclare the description name of the plugin.

upstream

Declares the upstream server url of the proxy. Only declare if the plugin is a proxy.

middlewares

Declares the middlewares of the plugin in a list format.

context_processors

Declares the context processors of the plugin in a list format too.

dependency

Declares the additional installed apps that this plugin depends on. This doesn't automatically install the python dependencies, only add to django apps.

password_validators

A list of functions to validate password in the moment it's set. This allows plugins to define their own password validators. For example if the proxied app requires the password to have at least one upper case character it should provide a password validator for that.

urls

include
: Declares the include urls.

prefix
: Declares the prefix for the url.

    - Atention: Any URL used in the plugins' settings should not be preceded by "/"

menu

These variables defines the menu title and links of the plugin.

menu_title
: Declares the menu title.

menu_urls

Declares the menu items and its links. This should be a tuple object with several colab_url elements. The colab_url_factory creates a factory for your links along with your namespace. The auth parameter indicates wether the link should only be displayed when the user is logged in. The `kwargs` parameter receives a dict, where the key `path` should be a path URL to the page. Remember that this path is a URL, therefore it should never be preceded by "/".

Example:

```
from colab.plugins.utils.menu import colab_url_factory

url = colab_url_factory('plugin_app_name')

menu_urls = (
   url(display=_('Profile'), viewname='profile', kwargs={'path': 'profile/'},␣
↪auth=True),
   url(display=_('Profile Two'), viewname='profile2', kwargs={'path': 'profile/2'},␣
↪auth=True),
)
```

Extra Template Folders

COLAB_TEMPLATES

Default () (Empty tuple)

Colab's extra template folders. Use it to add plugins template files, and remember to use the app hierarchy, e.g if your app name is example, then you should put your templates inside `COLAB_TEMPLATES/example`. You can also use it to overwrite the default templates, e.g. if you want to overwrite the default footer, you simply need to add a file named `footer.html` to the folder where `COLAB_TEMPLATES` points to.

Extra Static Folders

COLAB_STATIC

Default [] (Empty list)

Colab's extra static folders. Use it to add plugins static files. It's used the same way COLAB_TEMPLATES is. Use it to overwrite or add your own static files, such as CSS/JS files and/or images.

Settings

Paste

Social Networks

SOCIAL_NETWORK_ENABLED

Default False

When this variable is True, the social networks fields, like Facebook and Twitter, are added in user profile. By default, this fields are disabled.

Customization

Home Page

Menu

Templates

Verify Inactive User

ACCOUNT_VERIFICATION_TIME

Default timedelta(hours=48)

This variable will be used to remove inactive user. By default, this time is 48 hours.

CHAPTER 2

Plugin Developer Documentation

Getting Started

To start a new plugin, run the command:

```
$ colab-admin startplugin plugin_name [directory]
```

Where `plugin_name` is the name of your new plugin. And `directory`, which is optional, specifies the directory where the structure of your plugin will be created.

Signals

Implement signals in plugins is optional! You may follow this steps only if you want to communicate with another plugins.

In order to configure a plugin to able to listen and send signals using Colab signals structure, some steps are required:

- In the apps.py file it is necessary to declare a list variable containing all the signals that the plugin will dispatch. It is suggested to name the variable "registered_signals", but that nomenclature is not strictly necessary.
- It is also necessary to declare a variable containing the name of the plugin that will send the signal. It must be said that the name of the plugin cannot contain any special character, such as dot or comma. It is suggested to name the variable "short_name", but that nomenclature is not strictly necessary.
- Finally, the variable namespace should also be defined. This variable is the url namespace for django reverse.
- In order to actually register the signals, it is necessary to implement the method register_signal, which require the name of the plugin that is registering the signals and a list of signals to be registered as parameters. You must not call this method nowhere.
- In order to listen for a given signal, it is required to create a handling method. This method should be located at a file named tasks.py in the same directory as the plugins files. It also must be said that this method need to receive at least a **kwargs parameter. An example of a handling method can be seen below:

```
from colab.celery import app

@app.task(bind=True)
def handling_method(self, **kwargs):
    # DO SOMETHING
```

- With signals registered and handling method defined you must connect them. To do it you must call connect_signal passing signal name, sender and handling method as arguments. These should be implemented on plugin's apps.py. It must be said that the plugin app class must extend ColabPluginAppConfig. An example of this configuration can be seen below:

```
from colab.plugins.utils.apps import ColabPluginAppConfig
from colab.signals.signals import register_signal, connect_signal
from colab.plugins.PLUGIN.tasks import HANDLING_METHOD

class PluginApps(ColabPluginAppConfig):
     short_name = PLUGIN_NAME
     signals_list = [SIGNAL1, SIGNAL2]
     namespace = PLUGIN_NAMESPACE

     def registered_signal(self):
         register_signal(self.short_name, self.signals_list)

     def connect_signal(self):
         connect_signal(self.signals_list[0], self.short_name,
           HANDLING_METHOD)
         connect_signal(self.signals_list[1], self.short_name,
           HANDLING_METHOD)
```

- To send a broadcast signal you must call send method anywhere passing signal name and sender as arguments. If necessary you can pass another parameters in **kwargs. As you can see below:

```
from colab.signals.signals import send

send(signal_name, sender)
```

- If you want to run celery manually to make some tests, you should execute:

```
colab-admin celeryC worker --loglevel=debug
```

Search

In order to make some plugin model's searchable, it is necessary to create some files. First of all, it is necessary to create a directory named "search" inside the templates directory, that should be found on the plugin root directory.

Once the search folder exist, it is necessary to create a html file that will describe how a model item will be displayed on the colab search page. This file name must follow the pattern:

MODELNAME_search_preview.html

Where the MODELNAME should be the name of the model object that will be represented on the html file. In this template, you can set the following variables:

- `modified`: value in django datetime format.
- `modified_time`: if setted as True, it shows `modified` date and time.

- `author`: HTML code with a link of responsible for the result profile.
- `url`: link for the result location.
- `title`: title of the result.
- `description`: description of the result.
- `registered_in`: category of the result, as string.

To set variables, you have to load the set_var templatetag (`{% load set_var %}`) in your template, then you can set the variables using this syntax:

```
{% set 'variable_name' variable_value %}
```

If you don't want a variable to be showed, you just shouldn't set it.

These variables will be used in the below code:

```
{% load i18n tz highlight %}
{% block content %}
  <div class="row">
    <div class="col-md-12">
      <small>{{ modified|date:"d F Y"|default_if_none:"" }}
        {% if modified_time %}
          {% trans "at" %} {{result.modified|date:"h:m" }}
        {% endif %}
        {{author|safe|default_if_none:""}}
      </small><br>
      <h4><a href="{{url}}">
        {% if title %}
          {% highlight title with query %}
        {% endif %}
      </a></h4>
      <p>
        {% if description != "None" %}
          <a href="{{url}}">{% highlight description with query %}</a>
        {% endif %}
      </p>
      {% if registered_in %}
        <small class="colab-result-register">{% trans "Registred in" %}:
          <strong>{% trans registered_in %}</strong>
        </small>
      {% endif %}
    </div>
    <hr>
  </div>
{% endblock content %}
```

As you can see above, it is also possible to highlight the elements being searched.

To illustrate how to use this template base, see the following code:

```
{% extends "search-base.html" %}
{% load set_var %}
{% block content %}
  {% set 'title' result.title %}
  {% set 'modified' result.modified %}
  {% set 'url' result.url %}
  {% set 'registered_in' "Code" %}
  {% set 'description' result.description|default_if_none:" "|truncatechars:"140" %}
```

```
  {{ block.super }}
{% endblock content %}
```

And the follow HTML will be generated:

```
<div class="row">
  <div class="col-md-12">
    <small>24 October 2014
    </small><br>
    <h4><a href="/gitlab/softwarepublico/colab/merge_requests/1">
        Settings fix
    </a></h4>
    <p>
        <a href="/gitlab/softwarepublico/colab/merge_requests/1"> </a>
    </p>
      <small class="colab-result-register">Registred in:
        <strong>Code</strong>
      </small>
  </div>
  <hr>
</div>
```

If your search preview doesn't match the base template, you just don't have to extend it and make your own HTML.

Also a another file that must be created is the search_index.py one. This file must be placed at the plugin root directory. This file dictates how haystack will index the plugins models. If there is any doubt about how to create this file, it's possible to check the official haystack documentation that can be seen on the bellow link.

Guide to create a SearchIndexesFiles

It can also be seen in the guide above that an indexes directory should be created. This directory should be placed inside the search directory originally created in this tutorial. Inside this directory, create a txt file for each model that can be queried. Each of these files must contain the model fields that will be searched if no filter is applied. If there is any doubts about how to create these files, please check the Guide to create a SearchIndexesFiles.

Storing TimeStamp

TimeStamp is a parameter to control the last time a model was updated, you should use it when you want the data updated after a given time. To do that, the colab model (colab.plugins.models) has a TimeStampPlugin class, used to store all last updates from timestamp from all plugins. The class methods used to handle TimeStamp can be seen bellow:

```
update_timestamp(cls,class_name):  # allow store a current datetime.
get_last_updated_timestamp(cls,class_name):   # allow get a datetime with last␣
↪timestamp stored from class_name.
```

Example Usage:

```
from colab.plugins.models import TimeStampPlugin

class TestPlugin():

    def update_timestamp(self):
        TimeStampPlugin.update_timestamp('TestPlugin')
```

```
    def get_last_updated_timestamp(self):
       return TimeStampPlugin.get_last_updated_timestamp('TestPlugin')
```

Password Validation

Allows the plugin to define rules to set the password. The validators are functions which receive the password as only argument and if it doesn't match the desired rules raises a *ValidationError*. The message sent in the validation error will be displayed to the user in the HTML form.

Example:

```
## myplugin/password_validators.py

def has_uppercase_char(password):
    for char in password:
        if char.isupper():
            return

    raise ValidationError('Password must have at least one upper case char')

## /etc/colab/plugins.d/myplugin.py

password_validators = (
    'myplugin.password_validators.has_uppercase_char',
)
```

Username Validation

Allows the plugin to define rules to set the username. The validators are the same as the password validators ones. Therefore, they follow the same structure.

Example:

```
## myplugin/username_validators.py

def has_uppercase_char(username):
    for char in username:
        if char.isupper():
            return

    raise ValidationError('Username must have at least one upper case char')

## /etc/colab/plugins.d/myplugin.py

 username_validators = (
    'myplugin.username_validators.has_uppercase_char',
)
```

Blacklist

If you don't want a page to be accessed, you should add in your configuration file (/etc/colab/plugins.d) an array of regex strings named 'blacklist' that stands for the urls. The pages will then return a 403 error (forbidden).

Ex:

```
blacklist = [r'^dashboard$']
```

It also must be said that the full url that will be blocked is a combination of the plugin prefix and one of the elements of the blacklist array. For example, given a plugin with this configuration:

```
urls = {
    'include': 'colab_plugin.urls',
    'prefix':  '^plugin/',
    }

blacklist = [r'^feature$']
```

The actual url that will be blocked will them be: plugin/feature.

Change Header

If you want to change the header on your plugin pages, you should add in your configuration file (/etc/colab/plugins.d/) a boolean variable 'change_header', where True uses the slim header and False uses the default header. The default value of 'change_header' variable is False.

```
urls = {
    'include': 'colab_plugin.urls',
    'prefix':  '^plugin/',
    }

change_header = True
```

CHAPTER 3

Developer Documentation

Getting Started

Widgets

A widget is a piece of HTML that will be inserted in a specific spot in a page to render some view.

To create a new widget you need to extend the `Widget` class from `colab.widgets`. In the child class you can override the methods below, but it is not mandatory:

`get_header`
: This method should return the HTML code to be used in the page's head. So, it will extract the head content from the `content`.

`get_body`
: This method should return the HTML code to be used in the page's body. So, it will extract the body content from the `content`.

`generate_content`
: This method will set the `content` when the widget is requested by the `WidgetManager`. The `content` contains a HTML code that will be rendered in the target page.

The Widget class has the following attributes:

`identifier`
: The identifier has to be a unique string across the widgets.

`name`
: The widget name is the string that will be used to render in the view, if needed.

Example Widget:

```
from colab.widgets.widget_manager import Widget

class MyWidget(Widget):
    identifier = 'my_widget_id'
```

```
    name = 'My Widget'
    def generate_content(self, **kwargs):
        # process HTML content
        self.content = processed_content
```

To add the widget in a view check the Widgets section in User Documentation. To use a widget in the templates, you have to use the `import_widget` tag inside the `html` block. This way, the kwargs parameter will have a `context` key from the `html`. You can also set the variable that the widgets of an area will be imported. Or you can use the default name, which is `widgets_area_name`. For example, in the `profile` area the variable name is `widgets_profile`. This variable will be inserted directly in the page `context`.

```
{% load widgets_tag %}

{% block html %}
   {% import_widgets 'profile' %}
   {{ block.super }}
{% endblock %}

{# example of how to use #}
{% block head %}
  {{ block.super }}

  {% for widget in widgets_profile %}
    {{ widget.get_header }}
  {% endfor %}

{% endblock %}

{% block body %}
  {{ block.super }}

  {% for widget in widgets_profile %}
    {{ widget.get_body }}
  {% endfor %}

{% endblock %}
```

Warning: Warning! Remember to use the tag `{{ block.super }}` inside the html block. Otherwise, the page will appear blank.

Blacklist

The blacklist is an array of urls that the user cannot access directly. This variable holds an array of urls regex that must be blocked. This variable must be set on settings.py, exactly as the following example:

```
BLACKLIST = [r'^dashboard$']
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index