# Django Project Base

## version 1.3

May 17, 2021

# Welcome to Django Project Base's documentation!

## What is django-project-base?

We start with a project. Everything revolves around it: users, roles, permissions, tags, etc. Everything belongs to a project first, then to database. This project makes it easy to work on that premise. At the same time it integrates a few basic operations that you need in every project so that you don't have to do them over and over again.

This is a django library, based on django-rest-framework with django-allauth integration.

## Why django-project-base?

Functionalities provided:

- A base Project definition and editor for it. Extend as you like.
- User profile editor. Manage emails, confirmations, social connections
- Support for REST-based authentication / session creation
- Session / user caching for speed
- Project users editor. Invite users to project. Assign them into roles.
- Roles management & rights assignment.
- Tags editor & manager + support API for marking tagged items with their colours or icons

Index:

### *Installation*

Install the package:

```
pip install django-project-base
```

Extend the BaseProject & BaseProfile model:

```
# myapp/models.py
from django_project_base import BaseProject

class MyProject(BaseProject):
    # add any fields & methods you like here

class MyProfile(BaseProfile):
    # add any fields & methods you like here
```

Then also make sure your models are loaded instead of django-project-base models:

```
# myproject/settings.py

DJANGO_PROJECT_BASE_PROJECT_MODEL = 'myapp.MyProject'
DJANGO_PROJECT_BASE_PROFILE_MODEL = 'myapp.MyProfile'

# urls.py add
from django_project_base.router import django_project_base_urlpatterns
urlpatterns = [ ... ] + django_project_base_urlpatterns

Add to INSTALLED_APPS
 'rest_registration',
 'django_project_base',
 'drf_spectacular',

Add:
  REST_FRAMEWORK = {
 # YOUR SETTINGS
```

```
'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}
```

### *Warning*

This is important!!! You need to do the overriding before you create migrations. Migrating after default models had been created and used is a really hard and painful process. So make triple sure you don't deploy your application without first making sure the model you want to use is either your own or you are satisfied with our default implementation.

## *Settings*

```
DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES: {
    'project': {'value_name': 'current_project_slug', 'url_part': 'project-'},
    'language': {'value_name': 'current_language', 'url_part': 'language-'}
}
```

This setting defines dictionary of attribute names on request object. For e.g. project info is set on request object under propery current_project_slug. Language information is set on request objects under property current language. Is language or project is given in request path like: language-EN, then url_part settings is found and EN string is taken as language value.

```
DJANGO_PROJECT_BASE_SLUG_FIELD_NAME: 'slug'
```

When creating models with slug field they should be named with this setting value. This enables that we can use object slug instead of object pk when making api requests.

## *Javascript Client*

### *Usage*

Look at django_project_base/templates/index.html for examples.

**API Documentation**

Swagger UI is accessible on /schema/swagger-ui/ url by running example project.

**Translations**:

If you want to use your Django translations in your app include <script src="{% url 'javascript-catalog' %}"></script> in your html document header.

**Titlebar component integration example**

```
# define view function, put it in one of urls definition in urls.py
from django.shortcuts import render

def index_view(request):
   return render(request=request, template_name='template.html')
```

```
<!-- prepare html template template.html -->

 {% load static %}
 <!DOCTYPE html>
 <html lang="en">
 <head>
     <meta charset="UTF-8">
     <title>Titlebar component example</title>
     {# include django javascript catalog for internationalization #}
     <script src="{% url 'javascript-catalog' %}"></script>
     {# add bootstrap library with dependencies and font-awesome #}
```

```
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.2/css/all.min.css"
        crossorigin="anonymous">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.js" crossorigin
    <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.1/css/bootstra
        crossorigin="anonymous">
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" cross
    {# include django project base js lib and appropriate css #}
    <link href="{% static 'bootstrap_template.css' %}" rel="stylesheet" crossorigin="anonym
    <script src="{% static 'django-project-base.min.js' %}"></script>
</head>
<body>
    {# set div which will contain titlebar component #}
    <div id="titlebar-app" class="titlebar-app">
        {# use/render titlebar component #}
        <titlebar>
        </titlebar>
    </div>
    {# include vue inline template for titlebar component from folder coresponding to inclu
    {% include "bootstrap/titlebar.html" %}
    <script>
        // initialize titlebar component
        createApp('titlebar-app', titlebar);
    </script>
</body>
</html>
```

### For developers

For code formatting use .jshintrc file present in repository. Set tab size, ident, continuation ident in your editor to 2 places.

For JS development go to https://nodejs.org/en/ and install latest stable version of nodejs and npm. In project base directory run npm install. To run a development server run npm run dev (go to http://0.0.0.0:8080/). To generate a build run npm run build.

JS code is present in src directory. For web UI components library vuejs(https://vuejs.org/) is used. Components are built as Vue global components(https://vuejs.org/v2/guide/components.html) with x-templates. Templates are present in templates directory.

When developing webpack development server expects that service which provides data runs on host http://127.0.0.1:8000. This can be changed in webpack.config.js file. For running example django project prepare python environment and run (run in repository root):

- pip install -r requirements.txt (run in content root)
- python manage.py runserver

Try logging in with user “miha”, pass “mihamiha”.

## Authentication

### Impersonate user

Sometimes is useful if we can login into app as another user for debugging or help purposes. User change is supported via REST api calls or you can use userProfile component (django_project_base/templates/user-profile/bootstrap/template.html) which already integrates api functionality. Functionality is based on django-hijack package.

For determining which user can impersonate which user you can set your own logic. Example below:

```
# settings.py
 HIJACK_AUTHORIZATION_CHECK = 'app.utils.authorization_check'

# app.utils.py
 def authorization_check(hijacker, hijacked):
```

```
    """
    Checks if a user is authorized to hijack another user
    """
    if my_condition:
        return True
    else:
        return False
```

### User caching backend

To increase AUTH performance you can set a backend that caches users.

To enable User caching backend to add the following line to *AUTHENTICATION_BACKENDS* section in settings.py:

```
# myproject/settings.py

AUTHENTICATION_BACKENDS = (
    ...
    'django_project_base.base.auth_backends.UsersCachingBackend',  # cache users for auth to
    ...
)
```

User caching is not enabled for bulk updates by default, since Django doesn't call signal on .update() .bulk_update() or .delete(). Updating data with a query or running bulk update, without clearing cache for every object could potentially cause race conditions. Avoid it if possible, or take care of manually clearing the cache for the user.

Example for clearing cache after bulk update:

```
...
from django.core.cache import cache
from django_project_base.settings import DJANGO_USER_CACHE
...
# Bulk update multiple users. Give them superuser permission.
# If those users are logged in, they don't have permission until cache is cleared or they lo
UserProfile.objects.filter(username__in=['miha', 'janez']).update(is_superuser=True, is_staf

# After clearing users cache for those users will be able to work with additional permission
staff = UserProfile.objects.filter(username__in=['miha', 'janez'])
      for user in staff:
          cache.delete(DJANGO_USER_CACHE % user.id)
```

It is possible to add a clear cache option also for bulk updates if needed with a custom QuerySet manager. You can find example code below.

```
# models.py
...
from django.core.cache import cache
from django_project_base.settings import DJANGO_USER_CACHE
...
class ProfilesQuerySet(models.QuerySet):
    def update(self, **kwargs):
        for profile in self:
            cache.delete(DJANGO_USER_CACHE % profile.id)
        res = super(ProfilesQuerySet, self).update(**kwargs)
        return res

    def delete(self):
      for profile in self:
          cache.delete(DJANGO_USER_CACHE % profile.id)
      res = super(ProfilesQuerySet, self).delete()
      return res


class UserProfile(BaseProfile):
    """Use this only for enabling cache clear for bulk update"""
```

```
    objects = ProfilesQuerySet.as_manager()
...
```

## Tags

Django project base supports tags usage. See example implementation bellow.

```
class DemoProjectTag(BaseTag):
    content = models.CharField(max_length=20, null=True, blank=True)
    class Meta:
        verbose_name = "Tag"
        verbose_name_plural = "Tags"

class TaggedItemThrough(GenericTaggedItemBase):
    tag = models.ForeignKey(
        DemoProjectTag,
        on_delete=models.CASCADE,
        related_name="%(app_label)s_%(class)s_items",
    )

class Apartment(models.Model):
    number = fields.IntegerField()
    tags = TaggableManager(blank=True, through=TaggedItemThrough, related_name="apartment_ta

# Example code
from example.demo_django_base.models import DemoProjectTag
dt = DemoProjectTag.objects.create(name='color tag 20', color='#ff0000')

from example.demo_django_base.models import Apartment
a = Apartment.objects.create(number=1)
a.tags.add(dt)
a.tags.all()

<QuerySet [<DemoProjectTag: color tag 20>]>

# Get background svg for tags
DemoProjectTag.get_background_svg_for_tags(Apartment.objects.all().first().tags.all())
```

## Fields

### HEXColorField

Field with validator for color in hex format, currently used for setting background color for Tags.

## Middleware

***Project Middleware***

ProjectMiddleware: If you wan't to set current project which is selected to request object you can use ProjectMiddleware which should be placed to start of MIDDLEWARE list in settings.py. Middleware sets DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES setting dict values to request object. Default value for DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES setting is {'project': 'current_project_slug', 'language': 'current_language'}. This means request will have current_project_slug attribute which will have value set to current project slug and request will have current_language attribute which will have value set to current language set. If project or language cannnot be determined its value is set to None.

To set current project to ajax requests 'Current-Project' header should be used: 'Current-Project': 'current project slug'. Current slug can also be determined from request path. See DJANGO_PROJECT_BASE_PROJECT_DEFINED_URL_PART setting description in setting section.

```
# myproject/settings.py

MIDDLEWARE = [
```

```
 'django_project_base.base.middleware.UrlsVarsMiddleware',
 ...
 ]
```

# Modules

The page contains all information about Django Project Base modules:

## Notifications

### What is notifications module?

Notifications module will provide functionality to create and deliver notifications to users via channels like: email, websocket, push notification,.. Currently only maintenance notifications are implemented.

### Maintenance notifications

**Description**

When we have a planned server downtime to upgrade or some such, we need to somehow notifiy the users. But before maintenance occurs, the app itself must also notify the users that server will soon be down for maintenance. This notifications is presented to users 8 hours before planned downtime, 1 hour before planned downtime, 5 minutes before server is going offline.

In order to achieve that we can create a maintenance notification via REST api described in Swagger UI. If we have django project base titlebar UI component integrated into our web UI this component will display notifications for planned maintenance in above described intervals.

**Installation**

Add app to your installed apps.

```
# myproject/settings.py

INSTALLED_APPS = [
 ...
 'django_project_base.notifications',
 ]
```

Make sure you have django project base urls included:

```
# url.py

urlpatterns += django_project_base_urlpatterns
```

Run migrations:

```
python manage.py migrate
```

## Performance middleware

Performance middleware module is providing functionality to log and display the summary of the most time-consuming requests.

### Installation

To enable middleware add following line to projects settings.py

```
# myproject/settings.py

MIDDLEWARE = [
  ...
  'django_project_base.performance_middleware.middleware.profile_middleware.profile_middlewa
  ...
]
```

### *View*

Overview of current state is avialable on url *../app_debug/*

## *Example project*

You can find examples of most of the functionality of Django project base project in */example/* folder.

### *Run example project*

Run Python runserver from root directory of this project and visit url that is provided in command output.

```
$python manage.py runserver

...
Django version 3.1.8, using settings 'example.setup.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
...
```

### *Sample data*

**Users**

- **miha**:
  - username: miha
  - password: mihamiha
- **janez**:
  - username: janez
  - password: janezjanez