Django Project Base

version 0.1.15

January 07, 2022

Welcome to Django Project Base's documentation!

What is django-project-base?

This project removes the boilerplate associated with project / user handling: We start with a project. Everything revolves around it: users, roles, permissions, tags, etc. This project makes it easy to work on that premise: it provides foundations for user profiles, oauth authentication, permissions, projects, tagging, etc.

In order to take advantage of all this, just enable desired middleware and extend the models. This project will take care of the groundwork while you focus on your own project.

This is a django library, based on django-rest-framework with DynamicForms and Django REST Registration 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
- Various Vue components for visualising the above in browsers

Index:

Installation

Django project base

Install the package:

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

Extend the BaseProject & BaseProfile model:

Django project base uses Swapper https://pypi.org/project/swapper/, an unofficial API for Django swappable models. You need to override the Project and Profile models before you can use the library: there aren't any migrations available in the library itself. The library only declares properties it itself supports, but you have the option to extend them as you wish to fit your needs too.

```
# 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 swappable 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'

Add to INSTALLED_APPS
```

```
 'rest_registration',
 'django_project_base',
 'drf_spectacular',

Add:
  REST_FRAMEWORK = {
 # YOUR SETTINGS
 'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
 }
```

Append django project base urls:

```
# myproject/urls.py
urlpatterns = [
  ...
  path('', include('django_project_base.urls')),
  ...
]
```

Alternatively you can also choose to specify individual URLs. If that's the case, please refer to `django_project_base.urls.py` and use only the URLs you need.

> ***Note***
>
> The above general include also includes the Django javascript localisation catalog, so make sure you don't include it again.

There are some additional URLs available for the Django project base, like swagger or documentation. Appending those URLs is described in more details in respective chapters.

Dynamic Forms

Django project base is dependent on Dynamic Forms project https://github.com/velis74/DynamicForms

Read Dynamic Forms documentation for installation steps and more information about project.

You should add at least following code to your project, to enable Dynamic Forms.

```
# myproject/settings.py

REST_FRAMEWORK = {
...
      'DEFAULT_RENDERER_CLASSES': (
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
        'dynamicforms.renderers.TemplateHTMLRenderer',
        'dynamicforms.renderers.ComponentHTMLRenderer',
        'dynamicforms.renderers.ComponentDefRenderer',
    )
...
}
```

Environment setup

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

JS code is present in src subdirectory. For web UI components library vuejs(https://vuejs.org/) is used with single file components.

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 `vue.config.js` found in the same directory as `package.json`. For running example django project prepare python environment and run `{project base directory}`:

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

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

Activating features

User caching backend

Requires a settings.py AUTHENTICATION_BACKENDS setting. Optionally also a global cache server such as Redis. See Django CACHES setting.

Settings options - quick overview

DJANGO_PROJECT_BASE_PROJECT_MODEL

```
DJANGO_PROJECT_BASE_PROJECT_MODEL = 'myapp.MyProject'
```

Set swappable model for Django project base Project model. Read more in Django project base chapter.

DJANGO_PROJECT_BASE_PROFILE_MODEL

```
DJANGO_PROJECT_BASE_PROFILE_MODEL = 'myapp.MyProfile'
```

Set swappable model for Django project base Profile model. Read more in Django project base chapter.

DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES

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

A dictionary of attribute names on the request object. Read more in DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES chapter.

DJANGO_PROJECT_BASE_SLUG_FIELD_NAME

```
DJANGO_PROJECT_BASE_SLUG_FIELD_NAME: 'slug'
```

Read more in Project slug chapter.

MAINTENANCE_NOTIFICATIONS_CACHE_KEY

```
MAINTENANCE_NOTIFICATIONS_CACHE_KEY=""
```

Read more in Maintenance notifications chapter.

MAINTENANCE_NOTIFICATIONS_USE_CACHED_QUERYSET

```
MAINTENANCE_NOTIFICATIONS_USE_CACHED_QUERYSET=<bool>
```

Maintenance notifications use cached queryset. Default is True.

Read more in Maintenance notifications chapter.

USER_CACHE_KEY

```
USER_CACHE_KEY = 'django-user-{id}'
```

Key name for user caching background. Default value is ‘django-user-{id}’. Read more in User caching backend chapter.

CACHE_IMPERSONATE_USER

```
CACHE_IMPERSONATE_USER = 'impersonate-user-%d'
```

Cache key name for impersonate user. Default value is ‘impersonate-user-%d’. Read more in Impersonate user chapter.

PROFILE_REVERSE_FULL_NAME_ORDER

```
PROFILE_REVERSE_FULL_NAME_ORDER = (bool)
```

Read more in Profile reverse name order chapter.

DELETE_PROFILE_TIMEDELTA

```
DELETE_PROFILE_TIMEDELTA = 0
```

Value in days, when the automatic process should delete profile marked as for delete. Read more in Deleting profile chapter.

DOCUMENTATION_DIRECTORY

```
DOCUMENTATION_DIRECTORY='/docs/build/'
```

Path for documentation directory.

PROFILER_LONG_RUNNING_TASK_THRESHOLD

```
PROFILER_LONG_RUNNING_TASK_THRESHOLD = 1000
```

Define treshold in ms for profiling long running tasks. Read more in Performance profiler chapter.

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_tags")

# 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.UrlsVarsMiddleware',
 ...
 ]
```

Performance profiler

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

To enable middleware add following line to project files:

```
# myproject/settings.py

MIDDLEWARE = [
  ...
  'django_project_base.profiling.profile_middleware',
  ...
]


# myproject/urls.py
from django_project_base.profiling import app_debug_view

urlpatterns = [
path('app-debug/', app_debug_view, name='app-debug'),
...
]
```

Overview of current state is available on url *http://hostname/app-debug/*

Performance profiler can be used to profile any function as long as the function is triggered by input request.

Example below:

```
# func variable marks the function name which we want to profile during request
func = 'name_of_function_to_be_executed'
from django_project_base.profiling.middleware import ProfileRequest
# we set profiling path to function name instead of default request path used in profiling.middleware
ProfileRequest({'REQUEST_METHOD': 'GET', 'HTTP_HOST': '', 'QUERY_STRING': '', 'PATH_INFO': ''},
               None, (), {})._set_profiling_path(func, '')
# function is called
res = globals()[func](**parameters)
# function finishes and on request end(response) profiling data is logged and it can be then viewed in http://hostname/app-debug/ view
```

Modules

The page contains all information about Django Project Base modules:

Project

Project API is core part of Django project base.

Project slug

DJANGO_PROJECT_BASE_SLUG_FIELD_NAME

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. Default value is “slug”.

DJANGO_PROJECT_BASE_BASE_REQUEST_URL_VARIABLES

```
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 a dictionary of attribute names on the request object. E.g. project info is set on the request object under property 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.

Profile

Account / profile API.

Django project base uses multi-table inheritance together with abstract base classes to provide boilerplate user profile fields. The goal of our profile was to provide some social aspects as well as cover small communities where personal details like phone numbers are more commonly used as means of communication. Of course, any of the fields may be freely skipped with customisation.

Profile reverse name order

Settings option **PROFILE_REVERSE_FULL_NAME_ORDER** defines first_name, last_name order for readonly field *full_name*. Default order is *False* - “First Last”. Changing setting to true will reverse order to “Last First”.

Global setting can be also overrided with profile option reverse_full_name_order (bool).

Deleting profile

Super admins can either delete profile or mark it for deletion in future.

User cannot delete their profile, they can only mark it for deletion in future. After confirmation for deletion, their profile is marked for deletion, user is logged out and is not able to log in or use features that require logged in user.

Settings value **DELETE_PROFILE_TIMEDELTA** defines how far in future user profile will be actually deleted with automatic process. Value is set in days. The intent is in keeping user data in case they change their mind and re-register.

Existing profile table troubleshooting

You may find yourself in a pinch if your project already has a user profile table and it’s not linked to django.auth.User model using multi-model inheritance. Instead, you might have implemented it with a separate OneToOneField or even a ForeignKey. Even worse, if you linked all the user fields to this model and not the django.auth.User model.

You are SOL: migration will not be a matter of extending the model, but rather one of REPLACING the model. It is, however, only a 4-step (optionally 5-step) process in terms of migrations:

1. Declare the new user profile model, new foreign keys to the profile model in all tables where you link to your existing model. Basically you have duplicated all the fields and the model. run *makemigrations*.
2. Create a new *runPython* migration where you copy all the values from existing fields to new fields. This cannot be done in the first migration, you will just get an error running it.

a. if your references to previous profile model were to its own ID and not to django.auth.User model ID, you will have to also perform the translations between the two ID fields. Should be relatively easy in you migration code, something like:

```
# assumes you had a relation named "user" in your profile table
model.objects.update(**{
    field_name + '_new': Subquery(model.objects.filter(pk=OuterRef('pk')).values(field_name + '__user')[:1])
})
```

3. Delete all the old fields and model
4. Rename all the new fields and remove pre/postfixes. Optionally rename the new model as well, but don't forget to keep the database table name (*class Meta: db_table = 'module_model'*).
5. If you decided not to rename everything back to original names, you will need to replace all the references throughout your code. If you're not into DRY, you might consider renaming as a less painful option. Having tests wiull help A LOT here.

You will now end up with a new model that replaces your old one. Of course, the entire procedure is only worth it if you have code from project base you like and would like to take advantage of. Code such as user merging, maintenance, profile editor, etc. Regardless, it's a pain taking a bit of time to solve. On the plus side: it's an opportunity for a bit of a refactoring that's long overdue anyway :D Actually this was written as I (one of the authors of the library) was converting one of our oldest projects to the new system. I think I just despaired and moaned for a couple of days before actually doing it in about two hours (I decided NOT to rename the new model and took advantage of the refactoring opportunity)…

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',
 ]
```

Add django-project-base notifications urls:

```
# url.py

urlpatterns = [
    ......
    path('', include(notifications_router.urls)),
    ......
]
```

Run migrations:

```
python manage.py migrate
```

Authentication

Obtaining and maintaining sessions

We support two methods of maintaining session information for your client: cookie-based and header-based.

When you perform the account/login function, you can choose whether the function should return a session cookie or JSON with session id. Add parameter “return-type” with value “json” to login function parameters. This will return “sessionid” parameter in returned json instead of cookie. There is no CSRF when session is passed by the authorization header. See swagger documentation on login for further details.

If you choose the cookie, you will then need to supply the cookie(s) to all subsequent requests. Likewise, if you opt for session id as a variable, you will have to provide Authorization header to all subsequent requests.

The default uses cookies as those also add a CSRF cookie providing a bit more security. Use of JSON / header should only be preferred for clients without support for cookies, such as background maintenance / data exchange scripts.

Our modified SessionMiddleware only overrides Django’s as much to also accept the Authorization header and clears the session and CSRF cookies in the responses.

Activate project base accounts API endpoints

```
# myproject/urls.py

urlpatterns = [
  path('account/', include('django_project_base.account.urls')),
  ...
]
```

Session middleware

To enable project base’s SessionMiddleware, replace Django contrib SessionMiddleware with project base’s SessionMiddleware in projects settings.py file. This is only necessary if you intend to support the JSON method for login and keeping the session id.

```
# myproject/settings.py

MIDDLEWARE = [
...
'django_project_base.account.SessionMiddleware',
...]
```

Use of json session id in subsequent requests

When using the Authorisation header, use returned session api as token with token type “sessionid” and returned sessionid as credentials.

```
Authorization: sessionid <credentials>
```

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. Procedure is described in https://django-hijack.readthedocs.io/en/stable/configuration/ (See “Custom authorization function”) By default only superusers are allowed to hijack other users.

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 replace django.contrib.auth.backends.ModelBackend with the following line to *AUTHENTICATION_BACKENDS* section in settings.py:

```
# myproject/settings.py

AUTHENTICATION_BACKENDS = (
    ...
    'django_project_base.base.auth_backends.UsersCachingBackend',
    ...
)
```

User caching does not work on bulk updates as Django doesn't trigger signals on update(), bulk_update() or delete(). Bulk updating user profiles without manually clearing cache for them will create stale cache entries, so make sure you clear any such cache entries manually using the provided `user_cache_invalidate` function.

Example for clearing cache after bulk update:

```
...
from django.core.cache import cache
from django_project_base.base.auth_backends import user_cache_invalidate
...
# 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 log out and log in again.
UserProfile.objects.filter(username__in=['miha', 'janez'])\
    .update(is_superuser=True, is_staff=True)

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

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

```
# models.py
...
from django.core.cache import cache
from django_project_base.base.auth_backends import user_cache_invalidate
...
class ProfilesQuerySet(models.QuerySet):
    def update(self, **kwargs):
        for profile in self:
            user_cache_invalidate(profile)
        res = super(ProfilesQuerySet, self).update(**kwargs)
        return res

    def delete(self):
      for profile in self:
          user_cache_invalidate(profile)
```

```
        res = super(ProfilesQuerySet, self).delete()
        return res


class UserProfile(BaseProfile):
    """Use this only for enabling cache clear for bulk update"""
    objects = ProfilesQuerySet.as_manager()
...
```

Social auth integrations

Django Project Base offers easy-to-setup social authentication mechanism. Currently the following providers are supported:

- **Facebook**
    - provider identifier: facebook
- **Google**
    - provider identifier: google-oauth2
- **Twitter**
    - provider identifier: twitter
- **Microsoft**
    - provider identifier: microsoft-graph
- **Github**
    - provider identifier: github
- **Gitlab**
    - provider identifier: gitlab

OAuth providers require redirect URL which is called after the authentication process in Oauth flow.

Your redirect url is: [SCHEME]://[HOST]/account/social/complete/[PROVIDER IDENTIFIER]/

Information which settings are required for a social provider can be found at https://python-social-auth.readthedocs.io/en/latest/backends/index.html

For social authentication functionalities Python Social Auth library was used. Please checkout this documentation to make any custom changes.

Installation

Add app to your installed apps.

```
# myproject/settings.py

from django_project_base.accounts import ACCOUNT_APP_ID

INSTALLED_APPS = [
    ...
    'social_django',
    ACCOUNT_APP_ID,
    ...
 ]
```

Make sure you have django project base urls included:

```
 # url.py

 urlpatterns = [
   .....
   path('account/', include(accounts_router.urls)),
```

```
    path('account/social/', include('social_django.urls', namespace="social")),
    .....
]
```

Run migrations:

```
python manage.py migrate
```

Social login integration example - Google

To enable a social provider create an account at provider webpage and create an oauth app. For example for Google OAuth login visit https://console.developers.google.com/apis/credentials. Click + CREATE CREDENTIALS and select Oauth Client ID. Then create OAuth app with OAuth Consent screen.

Example value for Authorized JavaScript origins can be http://localhost:8080.

Example value for Authorized redirect URIs can be http://localhost:8080/account/social/complete/google-oauth2/.

To enable Google OAuth login add folowing to settings:

```
# myproject/settings.py
# enable google social login
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '*Client ID*'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '*Client secret*'
```

Translations

Currently translations in JS code, are done with Vue custom translations method.

It should be trivial to enable Django javascript-catalog, but it doesn't work correctly at the moment. It might change to correct Django javascript-catalog in future.

VUE components

Project base currently supports a few components for visualisation of the exposed APIs.

The components are designed to be customisable making it easy to replace any subcomponents with your own.

Here's the overview of the built-in components:

TitleBar component

The toolbar component provides the application toolbar containing page title, breadcrumbs, projects overview and account UI. Each of the subsections is fully customisable, but here's what the built-in components do:

TitleBar

Master component providing the grayed out area at the top of the page. It also provides visualisation for any messages the user needs to see, e.g. maintenance notices.

Composed of project image, page title, messages toast, breadcrumbs, projects overview and user profile menu.

..file:

django_project_base/src/components/bootstrap/titlebar.vue

Props

```
darkMode: { type: Boolean, default: false }, // dark mode on when true
projectlistComponent: { type: String, default: 'ProjectList' }, // specify your own globally registered component
userprofileComponent: { type: String, default: 'UserProfile' }, // specify your own globally registered component
breadcrumbsComponent: { type: String, default: 'Breadcrumbs' }, // specify your own globally registered component
loginVisible: { type: Boolean, default: true }, // if user is not logged in, should we show the login inputs
```

All the *Component parameters can be left out to use the provided components.

UserProfile

Handles user menu and associated UX for logging in, out, impersonation, maintaining social connections, credentials, etc.

You need to enable the */account/* URLs so that the component can actually do its work. See Authentication.

Vue single file components

You can add django_project_base as a js library to your package.json when developing Vue projects.

Notifications

If you want to use notifications django project base integrates vue-notification library. If you are not using django project base components you can add notifications by adding <Notification/> component in your App.vue or other component if you wish to use notifications functionality.

Examples

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"
     rel="stylesheet" crossorigin="anonymous">
   <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.js"
     crossorigin="anonymous">
   </script>
   <link
     href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.1/css/bootstrap.css"
     rel="stylesheet" crossorigin="anonymous">
   <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
     crossorigin="anonymous">
   </script>
   {# include django project base js lib and appropriate css #}
   <link href="{% static 'bootstrap_template.css' %}" rel="stylesheet"
     crossorigin="anonymous">
   <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 included css file #}
   {% include "bootstrap/titlebar.html" %}
   <script>
     // initialize titlebar component
     createApp('titlebar-app', titlebar);
   </script>
 </body>
 </html>
```

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.
...
```

Serve Sphinx documentation on localhost

Include documentation url to project urls.

```
# url.py

urlpatterns = [
   .....
   re_path(r'^docs-files/(?P<path>.*)$', documentation_view, {'document_root': DOCUMENTATION_DIRECTORY},
         name='docs-files'),
   .....
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

Sample data

Users

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

Swagger

Installation

To enable swagger gui, add following to urls.py

```
# my_project/urls.py
urlpatterns = [
  ...
  path('schema/', SpectacularAPIView.as_view(), name='schema'),
  path('schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema', ),
      name='swagger-ui'),
  ...
]
```

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

Open Api

Add following to settings.py

```
# myapp/settings.py
REST_FRAMEWORK = {
```

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