Real Estates web app with Django

Project report

by Imanuel Febie
student at BINUS University
Major: Computer Science
Course: Program Design Methods
Lecturer: Jude Martinez
2201835800
L1CC
Odd Semester
November 22, 2018

Abstract

For the final project of my Programming Design Methods course I decided to create a real estates web application. This report will introduce each component for this project and will attempt to describe my thought process during the development time.

Contents

1 Introduction

This report will describe the main objective of this project and how I have been able to build a data driven web application using Python's most popular web framework, Django. Besides learning how to fully use the benefits of Django I also tried to keep in mind how to build something that could be ready for production. Furthermore I will try to show my understanding of Django.

My real estates web applications allows property agents to register themself via their email address, after registering for the first time they will be asked to submit extra information about their company which can be usefull for website visitors. After this process they have access to a custom content management system to their needs. The content management system allows them to add, update and delete properties (houses and apartment units).

If an apartment doesn't exist yet, then this can then be added by what I have called a content manager. The content manager can verify registered agent, write blog posts and as stated before he can add new apartments.

The user interface of this project is designed by using a simple CSS framework called Bulma. The userinterface is minimalistic and I tried to keep it as straightforward as possible.

1.1 Project Structure

Django is a so called **MVT** web framework which makes it easier to make complex data-driven web applications in a relative short time. For each Django project you will be writing **Models**, **Views** and **Templates**. The models handle everything that has to do with data. A view in Django is the controller, it steers everything and is the bridge between our models and templates. The templates is where we write the User Interface and where Django also allows us to write Python logic.

2 Authentication & Registration

A Django project consist of one or more apps. Each app should be good at doing one thing only and do it good. For example: facebook could have an app called events if it was written with Django, which handles all the logic for events. Since my project is a real estates web app, the two most important apps are **accounts** and **properties**. My project ended up with the following apps:

```
+ root_folder
  + accounts
  + blog
  + core_app
  + pages
  + properties
  - manage.py
```

I will go through each app and explain the import parts of them. The first two apps we will discuss, is the core_app and accounts

2.1 Allowing Email Authentication

```
$ python manage.py startapp core_app
```

The first project app I focused on was the **core_app**, which handled the logic for authenticating a user object through an email address instead of Django's default behaviour, which is to use a username field.

```python
# core_app/managers.py
class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **kwargs):
        if not email:
            raise ValueError('Email is required')
        email = normalize_email(email)
        user = self.model(email=email, **kwargs)
        user.set_password(password)
        user.save()
        return user

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.set_default('is_staff', True)
        extra_fields.set_default('is_superuser', True)
        extra_fields.set_default('is_active', True)
        return self.create_user(email, password, **extra_fields)
```

According to Django's documentation the code above is one of the steps we need to do in order to allow authentication with an email address. It's probarly also imported to note that the behaviour of a Django Model is done by it's built-in Model Managers. A Manager is basically the interface through which database query operations are provided to Django models. This particular Model Manager is for the User object. By default it has the username as it's first argument after self. Besides the BaseUserManager I also had to make changes to the AbstractBaseUser class and as with BaseUserManager I have to inherit this class which than allows me to override or extends from it. Below is the code snippet from my project where I implemented the customization of the AbstractBaseUser.

```python
# core_app/models.py
from django.contrib.auth.models import AbstractBaseUser, PermissionMixin
from .managers import CustomUserManager

class CustomUser(AbstractBaseUser, PermissionMixin):
    email = models.EmailField(unique=True)
    is_agent = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False,
        _('staff status'),
        default=False,
        _help_text=('Designates whether the user can log into this admin site'),)
    is_active = models.BooleanField(_('active'), default=True,
        help_text=_('Designates whether this user is active'),)
    USERNAME_FIELD = 'email'
    objects = CustomUserManager()
```

Customizing the User object should always be done at the beginning of a project since doing this later can be troublesome. Especially when a drastic change like this has to be made.

For a framework that aims to allow developers to build web application in a short amount of time, I do find it a lot of work to just let me authenticate with an email address instead of a username. Also note that on line 6 I have added the is_agent boolean attribute. I will use this when I want to determine whether a User object is an property agent or not. For example, if I wanted to show certain HTML elements only to an Agent, inside my template I could do:

```
# some html template
{% if request.user.is_agent %}
    <h1>Hello World<\h1>
{% endif %}
```

Thanks to Django's built-in request context processor and I have extended the AbstractBaseUser with is_agent, I can do this check inside the template.

Next I had to decide if wanted to add the Agent model inside the models.py of the core_app or started a new app. I chose to start a new app named accounts which focused on the Agent and ContentManager models. With the core_app I only focused on changing the authentication behaviour of the built-in User object.

Django also comes with a settings.py module where all the configuration is being done. I have to tell Django that I have customized the AbstractBaseUser. To do so, I need to add the following line:

```
1 # realestates/settings.py
2 AUTH_USER_MODEL = 'core_app.CustomUser'.
```

I am now able to use the CustomUser inside this project.

2.2 Agent, ContentManager & VerificationDocument model

```
$ python manage.py startapp accounts
```

In the accounts app, I have written three Models. The Agent, ContentManager and the VerificationDocument model. The Agent and the ContentManager model both save information about them for their profile. To connect these models to the CustomUser, I can use Django OneToOneField like in the code snippet below:

```
# accounts/models.py
from django.conf import settings

def user_directory_path(instance, file):
    return 'logos/user_{}/{}'.format(instance.id, file)

class Agent(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    name = models.CharField(max_length=125)
    is_company = models.BooleanField(default=False)
    is_independent = models.BooleanField(default=False)
    logo = models.ImageField(upload_to=user_directory_path,
        default='logos/default.png')
    # ...
    is_verified = models.BooleanField(default=False)

class ContentManager(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    first_name = models.CharField(max_length=125)
    last_name = models.CharField(max_length=125)
    # ..
```

By importing settings I have access to the AUTH_USER_MODEL variable that I have declared before. Declaring it as the first argument for the OneToOneField field, will set the one-to-one relatioship with the User object. So each User can have **one** Agent or **ContentManager** object.

The same OneToOneField has been applied for the VerificationDocument. The idea of the VerificationDocument object is to let property agents verify themself of being a legitimate business by submitting a document. The verification is done by the content manager.

```
# accounts/models.py
class VerificationDocument(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    name = moels.CharField(max_length=20)
    document = models.FileField(upload_to='documents')
```

The user_directory_path function at the top is so each time an agent uploads a logo image, it will be put inside its own folder just for seperation.

2.3 Creating a User Object

Using Django forms, generic views and template engine we can let the user interact with the database. In order for a property agent to starting using the content management system they have to register first. Django has built-in authentication views and forms to handle this. But since I modified the AbstractBaseUser and need to let Django know I want the User object to be an agent by setting the is_agent boolean field to True. This required customizing the behaviour of the UserCreationForm. I haven written the form class for this inside the forms.py module of the core_app.

```
# accounts/forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.db import transaction

from core_app.models import CustomUser

class AgentRegistrationForm(UserCreationForm):

    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = ['email']

    @transaction.atomic
    def save(self):
        user = super().save(commit=False)
        user.is_agent = True
        user.save()
        return user
```

The UserCreationForm class has the username, password1 and password 2 fields built-in. And just like a Django model form I have to tell which model to use. In my case the CustomUser and also override the fields list and explicitly tell I have an 'email' field, because else it expects a 'username' field. The password1 and password2 well render without explicitly adding them inside the fields list because I am using the UserCreationForm.

Next I override the save method by telling Django that before saving the user to the database, I want to set the is_agent attribute to True. The transaction decorator was initialy used because I tried to combine the CustomUser fields and some fields from the Agent object inside one transaction. I let it stay there in case I wanted try this functionality again.

I have devided the steps of creating a user that is an agent into two steps. The first registration page askeds them for their email and to set a password. The UserCreationForm does the email and password verification. After this the user will be authenticated and redirected to the profile form page. Here they can enter their agency information like their business name, whether they're independent or a company, phonenumber and upload their logo. If a logo isn't provided then it will fallback to a default image. For the phonenumber field I used a Django third party package called **django-phonenumber-field**, which can be installed using pip or pipenv. This package does phonenumber verification so we know for sure the input by the user is a phonenumber.

2.3.1 The CreateView

The models and forms by themself can't do anything if there is no view for them and a template where users can actually interreact with the application. So inside the views.py of the accounts app I wrote the views for the former forms. Django comes with **generic class based views** so we as developers don't have to reinvent the wheel again. One of those generic views is the CreateView. The code for this looks as follows:

```
# accounts/views.py
from django.contrib.auth.mixins import LoginRequiredMixin
from django.shortcuts import redirect
from django.urls import reverse_lazy
from django.views.generic import CreateView

from core_app.models import CustomUser
from .forms import AgentRegistrationForm
from .models import Agent

class AgentRegistrationView(CreateView):
    model = CustomUser
    form_class = AgentRegistrationForm
    template_name = 'registration/register.html'

    def form_valid(self, form):
        user - forms.save()
        login(self.request, user)
        return redirect('registration_success')

class AgentProfileCreateView(LoginRequiredMixin, CreateView):
    model = Agent
    form_class = AgentProfileForm
    template_name = 'registration/registration_success.html'
    success_url = reverse_lazy('account:house_list')

    def form_valid(self, form):
        # assign user to requested user
        form.instance.user = self.request.user
        return super().form_valid(form)
```

First there are the nessacary imports. A CreateView can then be implemented by creating a class that inherits from it. We can then create an object by telling what model should be used and what form_class. If I didn't wrote a django form then I could use the fields attribute like inside class Meta of the UserCreationForm. Then I tell which template it should use by setting the template_name to the correct path. The form_valid function is a built-in method I override to tell it should redirect the user to the specified template name. The template name should exist or else it will throw an error. Naming a template or url path can be done inside the urls.py module where django searches for the right path using regular expressions. Usually you want to add the views for your app inside the urls.py of the app itself. But in my case I added them inside the main urls.py.

```python
# real_estates/urls.py
from django.contrib.auth.views import LoginView, LogoutView
from django.urls import path

from accounts.views import AgentRegistrationView, AgentProfileCreateView

urlpatterns = [
    # AUTH urls
    path('login/', LoginView.as_view(template_name='registration/login.html'),
        name='login'),
    path('logout/', LogoutView.as_view(),
        name='logout'),
    path('register/', AgentRegistrationView.as_view(),
        name='register'),
    path('register/success/', AgentProfileCreateView.as_view(),
        name='register_success'),
]
```

For each Django view there has to be a URLconfig. Above you can see the path for the costum views for registering a new user and creating a profile for the agent. As well, I imported the LoginView and the LogoutView. In my case it was not nessacary to write a login or logout view from scratch because Django already provided one for me that works for this project. Inside the as_view() method of the LoginView, I tell which template to use. This view already comes with a authentication form. The LogoutView can simply be called upon using a link with the url template tag provided with the name of the LogoutView inside the template as follow:

```
    <a href="{% url 'logout' %}">Logout</a>
```

By only importing the existing LoginView and tell which template to use, I can render the authentication form like below:

```
<!-- templates/registration/login.html snippet -->
<form method="post">
{% csrf_token %}
{{ form }} <!-- Will render the login form -->
<input type="submit" value="Login">
```

As long as the user object exist and the correct password is given, this form will work out of the box. The {{ form }} context is a naming convention Django uses for all it's built views with forms. So a form can be rendered the same way for the registration and profile creation views.

3 Designing Models

So far I have discussed the *core_app* and *accounts* app in general and explained how property agents can register, create their profile and login again. This chapter will discuss how I designed my models in general and how they affect the behaviour of the project. Most of the time when starting a new app, the first thing you will do is design the models.

3.1 Properties App and the Abstract Model

The properties app has the **Apartment**, **ApartmentUnit** and the **House** models. The ApartmentUnit and House model inherit from an abstract model I have made named **AbstractPropertyBase**. Inside the AbstractPropertyBase class I can put common information into the the house and apartment model. The abstract class itself will not have a database table but its fields will be added in the child classes. To create an abstract class set **abstract = True** inside the **Meta** class. And to use this class as the base, you simply inherit from it.

```
# properties/models.py
class AbstractPropertyBase(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    # auto_now_add=True will automatically set the time each time a
    # new object is created
    timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        abstract = True # makes the class abstract

class House(AbstractPropertyBase): # inherits from AbstractPropertyBase
    garden = models.BooleanField(default=True) # extra field for house
    ...

class Apartment(models.Model):
    name = models.CharField(max_length=125, unique=True)
    slug = models.SlugField(unique=True)
    floors = models.IntegerField()
    ...

class ApartmentUnit(AbstractPropertyBase): # inherits from AbstractPropertyBase
    apartment = models.ForeignKey(Apartment, related_name='units', on_delete=models.CASCADE)
    floor = models.IntegerField() # extra field for apartment unit
    ...
```

3.1.1 MoneyField

The **MoneyField** comes from a third party Django package which allows proper fields for currencies. I have used this field for three fields. The **deposit**, **rent_per_month** and **rent_per_year** which all are inside the AbstractPropertyBase class. The rent per year is something that the property agent does not need to worry about because it is auto filled each time a new property instance is saved.

```
    # code snippet
    class AbstractPropertyBase(models.Model):
    ...
    deposit = MoneyField(max_digits=14, decimal_places=2,
        default_currency='IDR')
    rent_per_month = MoneyField(max_digits=14, decimal_places=2,
        default_currency='IDR')
    rent_per_year = MoneyField(max_digits=14, decimal_places=2,
        default_currency='IDR')
    ...
    def save(self, *args, **kwargs):
        self.rent_per_year = self.rent_per_month * 12
        super().save(*args, **kwargs)
```

With MoneyField I can set the default_currency to **"IDR"** since this application in theory is targeting Jakarta.

3.1.2 Sorting models

For allmost all of my models in this project I like to have a default order without writing a queryset. The order is based on the time a new object is saved into the database. In order to have this behaviour I have **DateTimeField** which auto saves the time each time an object is created. I like to have the most recent object at the top and this can be done by putting **ordering = ['-timestamp']** inside the Meta class. the timestamp is the DateTimeField and by having the "-" in front of it reverses the order in the way I want.

```
    # code snippet
    ...
    class Meta:
        ordering = ['-timestamp']
```

3.1.3 Canonical URLS

A common design in Django is to have a canonical URL for model instances. The cano To have this I added the get_absolute_url () method that returns the canonical URL. This method can be defined in each model I wish to have a canonical URL. On the next page is the canonical URL for the apartment unit.

```
    from django.urls import reverse
    # code snippet
    ...
    def get_absolute_url(self), :
        return reverse('properties:unit_detail', kwargs={'slug': self.slug})
```

The reverse method takes in the URL name and additional arguments, in my case a keyword argument that points to the **slug** field of the apartment unit object. I also have this method for the House, Apartment and Article object. The function is exactly the same it just points the correct URL. The use of this method will become more clear in the next section [4].

3.2 Blog

A real real estate website would probarly also have a blog to post articles regarding the city, the country or tips about searching the ideal home. So the blog app is the only remaining app that has a model, the **Article** model. Articles can be written and managed by the ContentManager. The same concepts that has been discussed before like the canonical URL and ordering of the objects also applies for the Article model.

3.3 Pages

The pages app does not have any models. It only has one function based view that return the home page.

```
# pages/views.py
def home(request):
    return render(request, 'index.html')
```

The reason it has a homepage view is because I didn't know for sure if there was any context I wanted to give to the **index.html** template later.

In the next chapter I will discuss how all the models that have been created can be displayed to the outside world and how the property agent and content manager can manage them.

4 Django's Generic Views & Templates

In section 2.3.1 I've discussed how the user model works behind the scenes to create new user objects. For 90 percent of this project I have made use of the generic views. The generic views take away alot of reinventing design patterns in web development. The generic views I am talking about are the **CreateView**, **ListView**, **UpdateView** and **DeleteView**. Instead of explaining each generic view for each model I will discuss how each generic view can be used for every model.

4.1 Templates

In the view we controll who gets to see what and in the template we represent that by using html. Django's template engine allows to write some python logic inside the template. It also reduces repetition by allowing template inheritence and includes. Django also needs to know where to search for the templates. Inside the *settings.py* I can do that like this:

```
# settings.py
TEMPLATES_DIRS = os.path.join(BASE_DIR, 'templates')
..
TEMPLATES [
    ..
    'DIRS': [TEMPLATES_DIRS],
    ..
]
```

This way Django knows the templates are in the templates directory. Inside the templates directy I have individual directories for each app. Inside the templates directory it self I also have the *index.html*, *base.html*, *dashboard.html* and *overview.html*. The base is for where I all other templates inherit from which are for the regular website visitors. The dashboard is the base for the cms templates to inherit from.

4.2 Creating objects

```
# createview
from django.views.generic import CreateView

from .models import MODEL

class MODELCreateView(LoginRequiredMixin, CreateView):
    model = MODEL
    fields = ['field1', 'field2', 'field2']
    template_name = 'properties/house_create_form.html'
    # optiona: success_url = reverse_lazy('appname:url_name')

    def form_valid(self, form):
        form.instance.user = self.request.user
        return super().form_valid(form)
```

The code example above shows how each model can make use of the CreateView. The convention is usually as far as I know to have the model name followed by the name of the generic view. This view expects to have a model, the fields or form_class if a form class was created and it also needs to have a success url to redirect to. However, the success_url doesn't have to be explicitly set if there is a canonical URL for the model. And because all my models can only be created by authenticated users, I need to tell Django that the user fields needs to be assigned to the current authenticated user. This can be done by overriding the form_valid() method, access the user field of the model and assign th value to the user in session. The self.request.user can be used for this.

By default the CreateView has a context value for the template to render the form, which is {{ form }}. The generic views also has default templates naming conventions that we can use to create our templates. Unless I want to have my own template name the default is expected. To demonstrate I will use the House model to show how the CreateView can render a form in the template I assigned it to.

```
<!-- templates/properties/house_create_form.html -->
<form method="post">
{% csrf_token %}
<!--  render form -->
{{ form }}
<input type="submit" value="Submit">
</form>
```

The same logic applies for all the models when I want to create objects. In this case I didn't speficy where the view should be redirected to after creating the object, so it will go to its default behaviour and redirect to the canonical URL. The conical URL is the url of a single object.

4.3 Retrieving objects

To show the list of objects that have been created I have used the **ListView**. When no modification are needed you only need to write two lines of code. I am going to continue to use the House model as the example as we continue on.

```
1 from django.views.generic import ListView
2
3 from ..models import House
4
5 class HouseListView(ListView):
6     # only retrieve the available objects
7     queryset = House.objects.filter(available=True)
```

To tell which model I want this view to use I can either assign it to the **model** attribute or as in this case to the **queryset** attribute, because I only want to retrieve the houses that are set to available by the property agent. I also do this

for the ApartmentUnit model. For all the other models I can simply assign the model as I have done in the CreateView I showed before. The ListView expects a template file named "**model_list.html**". So for the House model this will be; "**house_list.html**".

Inside the template I can use a Python for loop to iterate over each object inside the the database and display them. The default context for the queryset is {{ modelname_list }}. So inside the template it works like this:

```
<!-- templates/properties/house_list.html -->
{% extends 'base.html' %}
..
{% block content %}
{% for object in house_list %}
    <h1){{ object.title }}</1>
    <small>{{ object.timestamp|date:'d M Y' }} </small>
    {{ object.description|linebreaks }}
{% endfor %}
{% endblock %}
```

Using dot notation I can access the attribute I want to display just like with regular Python.

4.3.1 Filter the retrieved objects per user

When the property agent authenticates he gets redirected to his personal content management system. To have an overview of his own properties I can filter to the **self.request.user** like below:

```
class MyHouseListView(LoginRequiredMixin, ListView):
    model = House
    template_name = 'properties/agent_house_list.html'

    def get_queryset(self):
        "override queryset
        retun House.objects.filter(owner=self.request.user)
```

Since I need to assign it to the owner instance of the model and therefor use *self*, I need to override the get_queryset() method, because inside the class I can't access the instance. So I can't use the queryset attribute as the example before. I also assign a different template for this view since the default expected template is already in use. I follow this same design pattern for the other methods where I need to filter the objects by the user they belong to.

4.3.2 Retreiving a single model instance

Besides displaying all the objects that exist You also would like to know the details of a for example one single apartment unit or a house. Instead retrieving all of them with the ListView, Django's has a DetailView that does exactly this, retreiving one single object. We can use the object's primary key or slug

to retrieve it from the database. I like to use the slug for this. A DetailView just takes 2 lines of code.

```
from django.views.generic import DetailView
..
class HouseDetailView(DetailView):
    model = House
```

The DetailView is where the canonical URL is send to which I have discusses in section 3.1.3. The url needs to have the slug in the path and can be done like this:

```
# ../urls.py
urlpatterns = [
    ..
    path('house/<slug:slug>/', HouseDetailView.as_view(), name='house_detail'),
]
```

End to display this objects information to the user on screen the DetailView gives {{ object }} as the context for the queryset, which can be used inside the template of the DetailView. So for example inside the template I can use {{ object.title }} directly to display the title of a house or apartment unit. As with all the generic view it expects a default template name if one is not explicitly being set. For the DetailView it expects: *modelname_detail.html*.

4.4 Updating objects

The **UpdateView** is almost the same as the CreateView. I assign a model to the class and tell which form class or fields it should render. The use of a creating a form class is not needed as you can see when using these generic views. Unless I want to customize what the form needs to do.

Though similar, the difference with the CreateView is that this view needs to know which existing instance it has to update. So in the request this view wants to know the *primary key* or *slug field* or both. For the apartment, apartment unit, article, agent and house I have added the **slug = models.SlugField()** into the model. A slug is how the url looks like in the browser. I like to use slug more than using the primary key since this looks better. In the UpdateView we don't have to write this logic ourself anymore, however we do need to pass the slug into the url pattern for this view.

```
from django.views.generic import UpdateView

class HouseUpdateView(LoginRequiredMixin, UpdateView):
    model = House
    fields = ['title', ...]
    template_name = 'properties/house_update_form.html'

    def get_queryset(self, form):
```

```
        form.instance.user = self.request.user
        return super().form_valid(form)
```

The url:

```
    urlpatterns = [
        ...
        path('house/<slug:slug>/update/', HouseUpdateView.as_view(),
            name='update_house'),
        ...
    ]
```

For the UpdateView I also specify which template to use. Rendering the form inside the template works the same as with the CreateView. By using the {{ form }} context. After a successfull update it will redirect to the canonical URL.

4.5 Deleting objects

The last generic view is the DeleteView. Like the UpdateView it needs to know which instance it needs to work with and it also by default identifies a model instance using the primary key or slug. Like with UpdateView I like to use slug.

```
from django.urls import reverse_lazy
from django.views.generic import DeleteView
..
class HouseDeleteView(LoginRequiredMixin, DeleteView):
    model = House
    success_url = 'account:house_list'
..
```

The url also looks the same as the UpdateView and DeleteView.

```
    ..
    path('house/<slug:slug>/delete/', HouseDeleteView.as_view(), name='delete_view'),
    ..
```

If a user desires to delete an object, the DeleteView will redirected him to a page where the action has to be confirmed. The template it looks for by default is *model_confirm_delete.html*. Inside this template we need to have a html form. This form however has no fields to be rendered so we only need to have a submit button that confirms the delete.

4.5.1 Confirm Delete Template

```
<!-- templates/properties/house_confirm_delete.html snippet -->
<form method="post">
{% csrf_token %}
<p>Are you sure you want to delete {{ object.title }}</p>
<input type="submit" value="Confirm">
</form>
```

5 Custom Context Processor & Template Tags