

Django

Creando una aplicación con django:

1. Creamos un entorno virtual
2. Instalamos django
3. Creamos una aplicación

```
pip install django
```

```
django-admin startproject project_name
```

APPS and PROJECTS

Hemos creado nuestro proyecto, con el comando anterior, pero ahora se nos crea también lo que se llama "aplicación", dentro del proyecto.

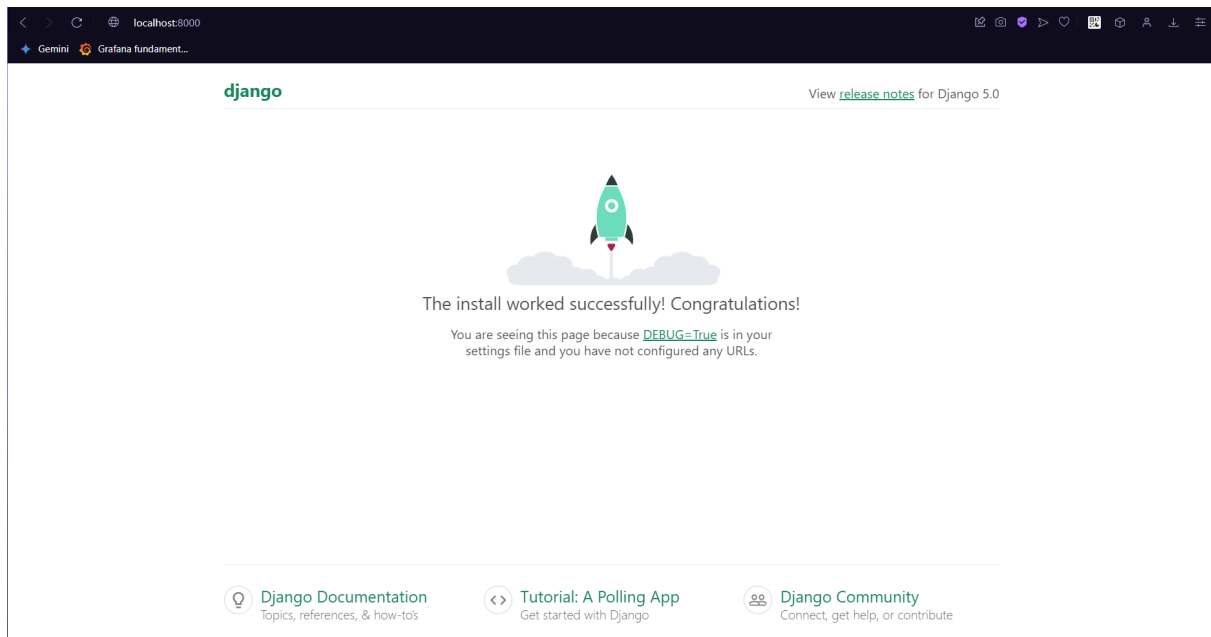
Nuestro proyecto "taskmate", tiene dentro de su carpeta otra carpeta "taskmate" que es la aplicación. Si queremos añadir otra aplicación a el proyecto (e.g. un blog) se añadiría una aplicación dentro del proyecto.

Para lanzar nuestra aplicación tenemos que dentro de la carpeta del proyecto "taskmate":

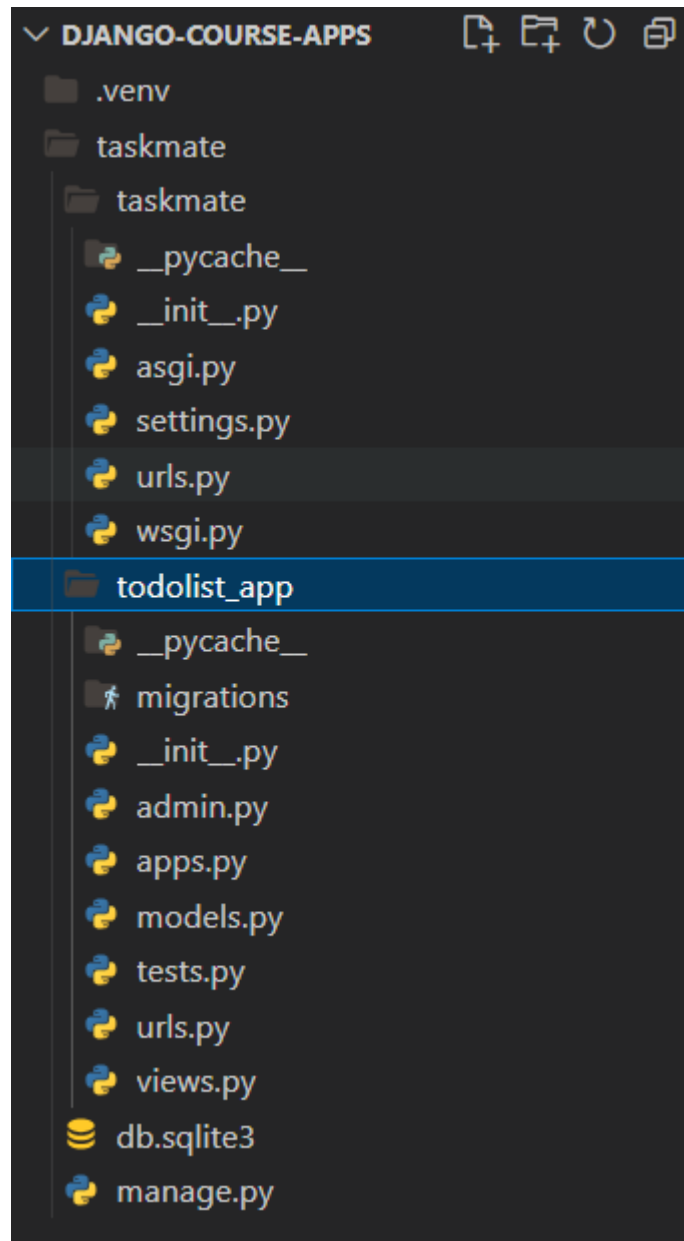
```
python manage.py runserver
```

Pero nos salen warnings de las migraciones, para arreglarlo hacemos estos dos comandos:

```
python manage.py makemigrations  
python manage.py migrate
```



Dentro de nuestra aplicación “taskmate” tenemos un archivo settings.py, que controla toda la aplicación, base de datos, modo de depuración (DEBUG = TRUE/FALSE)



Hemos creado nuestra primera “sección”, nuestra primera “aplicación” del proyecto, vemos que la *main* app podría ser **taskmate** que vemos que tiene el archivo `settings.py`.

Los pasos para crear la aplicación son los siguientes, como siempre nos colocamos en el directorio del *proyecto* .

```
python manage.py startapp todolist_app
```

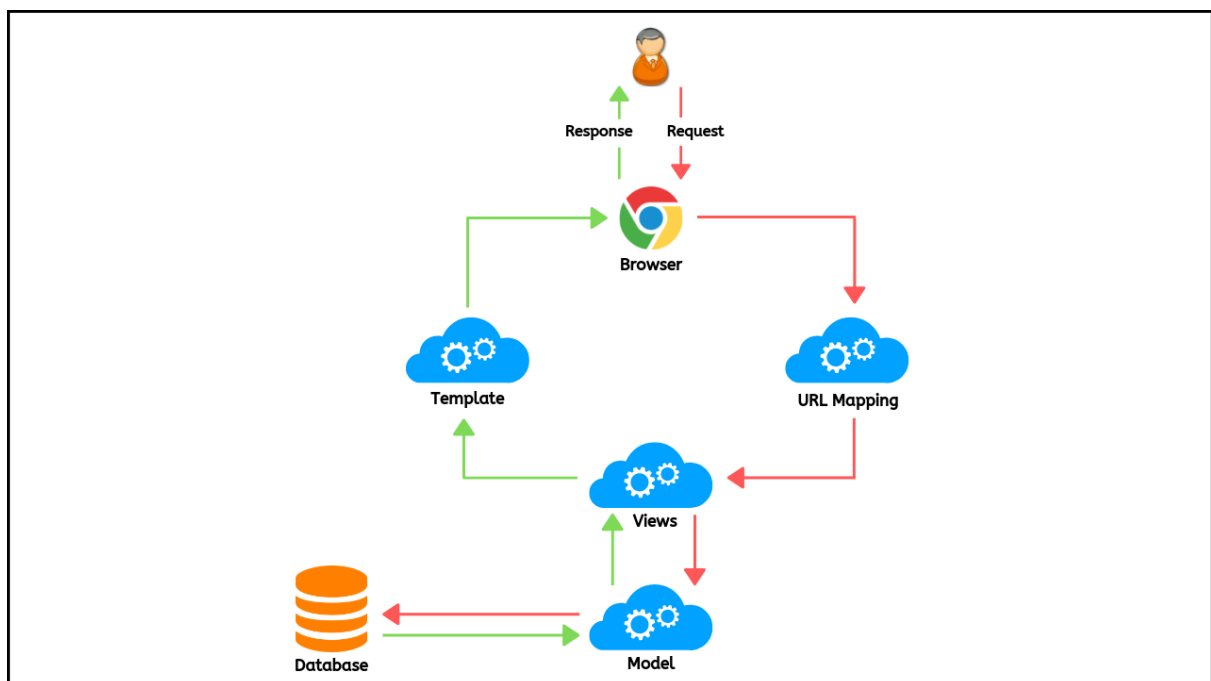
Después de esto se nos genera todo lo que vemos dentro de `todolist_app`, excepto el archivo **`urls.py`**, ¡que siempre crearemos!

No nos olvidemos de entrar en `settings.py` y añadir la aplicación:

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'todolist_app', # Add the app to the installed apps
]
```

Routing y workflow



En este momento, tenemos creada la aplicación principal, la que se encargará de redirigir todo lo que encontremos en nuestra aplicación (**taskmate**):

Vamos a crear nuestra primera *vista* que estará gestionada en nuestra app **todolist_app**.

Como vemos en el workflow, para llegar a una vista, tenemos que tener una URL mapeada, por tanto:

1. urls.py (en taskmate, nuestra aplicación principal) añadimos una ruta nueva:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('task/', include('todolist_app.urls'))
]

# La ruta /admin/ ya está configurada, y aquí simplemente hacemos
# usuario busque como localhost:8000/admin/ se deriva a las URLs
# pues hacemos lo mismo para las tareas, cualquier /task/ mirará
```

2. urls.py (en todolist_app)

```
from django.urls import path
from todolist_app import views

urlpatterns = [
    path('', views.todolist, name='todolist'),
]

# la homepage localhost:8000/task/ mirará en estas urls y verá
# a la vista todolist
```

3. creamos la vista a la que lo vinculamos en [todolist_app/views.py](#)

```
UNA VISTA PUEDE DEVOLVER LO QUE SEA! HTML, RESPUESTA HTTP, Y
COMUNICARSE CON LOS MODELOS (BASE DE DATOS) o DEVOLVER TEMPLATES

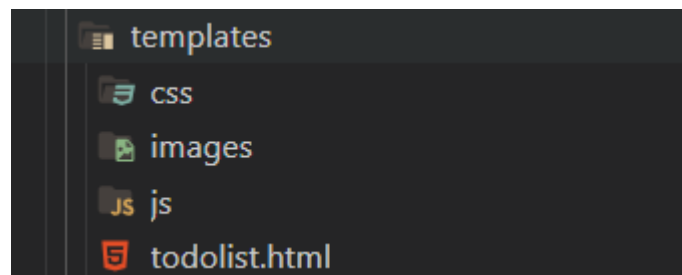
from django.http import HttpResponse
from django.shortcuts import render

# Create your views here.
```

```
def todoclist(request):  
    return HttpResponse("Bienvenido a la lista de tareas")
```



Si queremos vincularlo a un template, de una app específica, deberíamos primero crear la estructura de carpetas de los templates, en este caso únicamente estamos usando HTML CSS y JS, por tanto dentro del directorio de la aplicación todoclist_app creamos lo siguiente:



Ahora vinculamos en las vistas de nuestra aplicación dicho HTML.

```
from django.http import HttpResponse  
from django.shortcuts import render  
  
# Create your views here.  
  
def todoclist(request):  
    return render(request, 'todoclist.html', {})  
  
# Aprenderé mas tarde, pero entiendo que el {} es el contexto
```

DISCLAIMER: Para los elementos comunes de una web, por ejemplo una navbar, o un footer... todo iria a una carpeta commons, a nivel raíz, por encima

de cualquier app, pero lo veremos más tarde

Uso del Contexto y Jinja.

Sí, Django utiliza su propio motor de plantillas que tiene una sintaxis muy similar a Jinja. No necesitas instalar Jinja por separado cuando trabajas con Django, ya que el motor de plantillas de Django viene integrado y está listo para usar desde el momento en que creas una aplicación Django.

Aquí hay un resumen de cómo funciona el motor de plantillas de Django:

1. **Creación de la plantilla:** Escribes tus archivos HTML con la sintaxis de plantillas de Django, utilizando `{{ }}` para variables y `{% %}` para etiquetas de control (como bucles y condicionales).
2. **Contexto en la vista:** En tu vista de Django, pasas un diccionario de contexto a la plantilla. Este diccionario contiene las variables que deseas renderizar en la plantilla.
3. **Renderización de la plantilla:** Django toma la plantilla y el contexto, y genera el HTML final que se envía al navegador del usuario.

Efectivamente, usando el contexto en la vista, podemos pasarle valores a la plantilla y usarlos para que se renderice.

```
from django.http import HttpResponse
from django.shortcuts import render

# Create your views here.

def todoclist(request):
    return render(request, 'todolist.html', {'welcome_text':
```

Y en todoclist.html:

```
<div class="container">
    <h1>{{ welcome_text }}</h1>
</div>
```

ESTAMOS USANDO BOOTSTRAP!

HERENCIA DEL HTML y JINJA

De lo que nos hemos dado cuenta es que queremos partes de la app que sean iguales, por tanto crearemos un base.html que contiene todo aquello que es igual, en este caso la navbar.

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to=
    />

    <!-- Bootstrap CSS -->
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha
      rel="stylesheet"
      integrity="sha384-KK94CHFLLe+nY2dmCWGMq91rCGa5gtU4mk92H
      crossorigin="anonymous"
    />

    {% block title %}
    {% endblock title %}
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">
          <img src= "{% static 'images/logo.png' %}" alt= "Ta
          </img>
        </a>
        <button
          class="navbar-toggler"
          type="button"
          data-bs-toggle="collapse"
```



```

        data-bs-target="#navbarNav"
        aria-controls="navbarNav"
        aria-expanded="false"
        aria-label="Toggle navigation"
    >
        <span class="navbar-toggler-icon"></span>
    </button>
<div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link active" aria-current="page"
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'todolist' %}">
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'contact' %}">C
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'about' %}">Abo
        </li>
    </ul>
</div>
</div>
</nav>

{% block content %}
{% endblock content %}

<!-- Optional JavaScript; choose one of the two! -->
<!-- jQuery and Bootstrap Bundle (includes Popper) -->
<script
    src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha
    integrity="sha384-ENjdO4Dr2bkBIFxQpeoTz1HIcje39Wm4jDKdf
    crossorigin="anonymous"
></script>
</body>
</html>

```

Haciendo uso de Jinja nos damos cuenta de varias cosas.

1. Anotaciones del tipo `{%block %}`: Estas hacen referencia a "futuras" plantillas HTML que beberán de este `base.html` entonces ahora simplemente en otros archivos HTML, haciendo referencia a este bloque modificamos lo que queremos que cambie, por ejemplo el `content` o el `title`:

(Por ejemplo `todolist.html`)

```
{% extends "base.html" %}
{% block title %}
<title>
    Todo List - Taskmate
</title>
{% endblock title %}

{% block content %}
    <div class="container">
        <h1> {{welcome_text}}</h1>
    </div>

{% endblock content %}
```

2. Hacemos uso de las URLs que hemos definido en `urls.py`, que venia dado por el `name=`

```
from django.urls import path
from todolist_app import views

urlpatterns = [
    path('', views.todolist, name='todolist'),
    path('contact-us', views.contact, name='contact'),
    path('about-us', views.about, name='about'),
]
```

Cambiamos el nombre del path, simplemente poniendo con la anotación la URL, estamos a salvo

3. Por último el uso de ficheros static, para esto hemos creado una carpeta static a nivel raíz, que va a tener todo el CSS, JS e imagenes, para ellos simplemente la creamos y tenemos que cargarlo en settings.py (de la app principal, taskmate)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.0/howto/static-files/

STATIC_URL = 'static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')] # Add t
```

y cuando queramos usarlo no nos olvidamos de:

```
{% load static %}
<div class="container-fluid">
    <a class="navbar-brand" href="#">
        <img src= "{% static 'images/logo.png' %}" alt= "Ta
    </img>
    </a>
```

TODO GRACIAS A JINJA!

MODELS and ADMIN

Primero de todo hemos creado un super usuario para acceder al panel de administrador (/admin)

Sobre la carpeta de nuestro proyecto:

```
python manage.py createsuperuser
```

Ahora al lanzar el servidor y acceder a localhost:8000/admin podemos entrar en el panel de admin y loguearnos y controlar los modelos

Nuestro primer modelo (TaskList)

1. models.py de todolist_app

```

from django.db import models

# Create your models here.

class TaskList(models.Model):
    task = models.CharField(max_length=300)
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.task + "-" + str(self.done)

```

Para consultar qué tipo de dato le pongo a cada parámetro puedo consultar aquí:

<https://www.webforefront.com/django/modeldatatypesandvalidation.html>

Ya tienen las validaciones incluidas! Es como las @ de Spring?

2. Si quiero que se muestre en el panel de administrador en admin.py.

```

from django.contrib import admin
from todoapp.models import TaskList
# Register your models here.

admin.site.register(TaskList)

```

Una vez hecho todo esto, tenemos que hacer las migraciones, para que el modelo que hemos creado se cree como si fuese una *entidad* en base de datos, se crea automáticamente un id también y se pone como clave primaria

```

python manage.py makemigrations
python manage.py migrate

```

Y volvemos a lanzar el servidor

Se genera algo así al migrar:

```

# Generated by Django 5.0.7 on 2024-08-02 22:27

from django.db import migrations, models

```

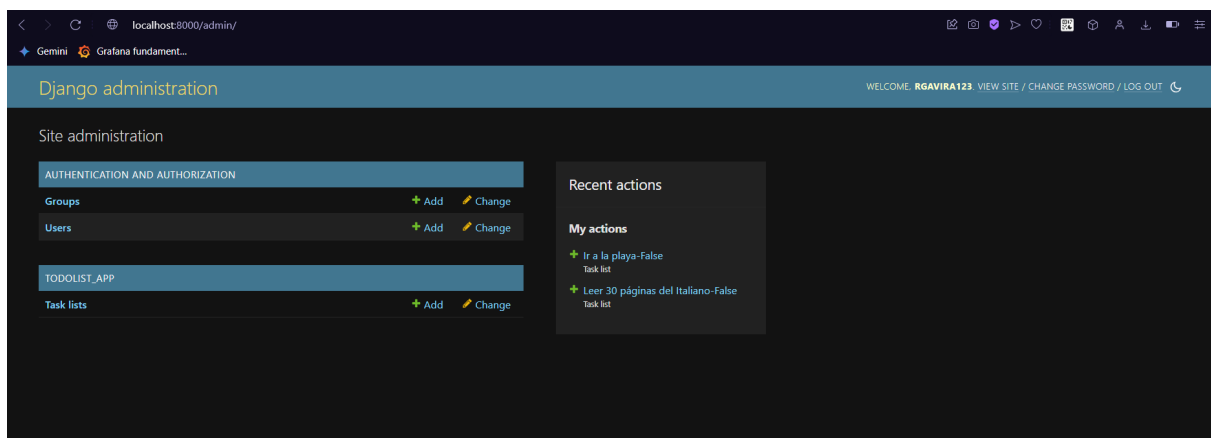
```
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='TaskList',
            fields=[
                ('id', models.BigAutoField(auto_created=True,
                ('task', models.CharField(max_length=300)),
                ('done', models.BooleanField(default=False))),
            ],
        ),
    ]
```

En este momento debo admitir que no sé donde se está guardando todo, pero ahora es curioso porque desde el panel de administrador puedo crear estas "entidades" y persisten:



El archivo de db.sqlite3 tiene contenido, bastante raro, imagino que irá por ahí la cosa.

FETCHING DATA from DB

En este punto es un poco "dudoso", no estamos accediendo a los objetos como estamos acostumbrados, simplemente al crear un modelo e ir guardando los datos podemos acceder a ellos con diferentes métodos.

Primero de todo tener claro la arquitectura MVT, hemos tocado el modelo (base de datos), nos hemos ido a la vista (lógica de negocio).

En las vistas ahora queremos llevar a la plantilla, al template todos los datos del modelo para trabajar con ellos allí:

En la vista asociada a la plantilla todolist.html:

```
def todolist(request):
    all_tasks = TaskList.objects.all # Get all the tasks from

    return render(request, 'todolist.html', {'all_tasks': all_
```

Vemos como hemos accedido a todos los elementos de TaskList con ese método.

Ahora en la plantilla todolist.html

```
{% extends "base.html" %}
{% block title %}
<title>
    Todo List - Taskmate
</title>
{% endblock title %}

{% block content %}
    <div class="container">
        {{all_tasks}}
    </div>

{% endblock content %}
```

Accedemos a todos con Jinja, porque se ha pasado como contexto de la aplicación, el resultado es simplemente para que visualmente veamos lo que estamos pasando, cabe recalcar que lo que estamos pasando tiene esta forma

<QuerySet [<TaskList: Leer 30 páginas del Italiano - False>, <TaskList: Ir a la playa - False>]>

Vemos que muestra la TaskList (cada item) y después lo que definimos el metodo str en el modelo, pero esto es un toString normal y corriente, realmente podemos acceder a cada elemento con un bucle for, usando Jinja, y creamos una tabla para cada tarea.

También podemos hacer ifs, al final estamos añadiendo código, como cuando en react hacemos {} para añadir código e iterar por cosas.

```
{% extends "base.html" %}
{% block title %}
<title>
    Todo List - Taskmate
</title>
{% endblock title %}

{% block content %}
<div class="container">
</div>
<table class="table table-bordered">
<thead class="thead-dark">
<tr>
<th scope="col">Task</th>
<th scope="col">Done</th>
<th scope="col">Edit</th>
<th scope="col">Delete</th>
</tr>
</thead>
<tbody>
{% for obj in all_tasks%}
    {% if obj.done%}
<tr class="table-success">
<td> {{obj.task}}</td>
<td> Completed</td>
<td> Edit </td>
<td> Delete </td>
</tr>
```

```

        {% else %}
        <tr>
            <td> {{obj.task}}</td>
            <td> To be done</td>
            <td> Edit </td>
            <td> Delete </td>
        </tr>
        {% endif %}
    {% endfor %}
</tbody>
</table>

</div>

{% endblock content %}

```

Los "atributos" del objeto que iteramos los podemos ver en las migrations, son los fields, y accedemos como si fuera un objeto

Creating Forms

Forms are meant to transfer data between the template and the database. This can be done through the views, it is served as a request, and the form's method can be changed to POST and in the respective view we can filter this to do as we please:

For this to be done, we need to modify our HTML (template), our view and create a forms.py that will be as some kind of middleware between these two:

1. Modify the form in our template todolist.html

```

<form method ="POST">
    {% csrf_token %}
    <div class="form-group">
        <input type="text" class="form-control" name="tas
    </div>

    <div class="mt-2">
        <button type="submit" class="btn btn-primary">Add

```



```
</div>
</form>
```

Note that in our input we have the attribute name, this attribute has to have the same name as the field we are referring to in the following file.

It can be seen that there is no 'done' input, this is because we have it set as False by default, so there is no need to have an input for that field.

2. Create our `forms.py` in our app

```
from django import forms
from todolist_app.models import TaskList

class TaskForm(forms.ModelForm):
    class Meta:
        model = TaskList
        fields = ['task', 'done']
```

In this form we refer to our previous model, the one we are "modifying"

3. Modify our view to receive all this data through the form and save our new entity:

```
def todolist(request):

    if request.method == "POST":
        form = TaskForm(request.POST or None)
        if form.is_valid():
            form.save()
            return redirect('todolist')

    else:
        all_tasks = TaskList.objects.all() # Get all the tasks

        return render(request, 'todolist.html', {'all_tasks':
```

Mensajes

Hemos podido pasar mensajes de la vista a la plantilla haciendo uso de `django.contrib.messages`

Simplemente en la view que se encarga de gestionar nuestra todomlist:

```
messages.success(request, ("New Task Added!"))
```

Implementando CRUD!

Vamos a trabajar de una forma diferente a la que estamos acostumbrados usando una API REST, con la api simplemente le pedíamos al cliente que al pulsar en un Delete se hiciera una llamada al servidor de backend pasandole los datos correspondientes.

Como django es capaz de vincular una URL a una *view* (que se encarga de la lógica de negocio digamos), asociaremos una URL para que elimine, actualice...

Delete

En `urls.py`:

```
path('delete/<task_id>', views.delete_task, name='delete_t
```

En `views.py`:

```
def delete_task(request, task_id):
    task = TaskList.objects.get(pk=task_id)
    task.delete()

    return redirect('todolist')
```

Es bastante sencillo, con `<task_id>` estamos indicando que vamos a pasar un parámetro a dicha URL, y el `name` que le damos es el nombre con el que gracias a Jinja accederemos después a dicha URL

En `todolist.html` modificamos el campo de Delete:

```
<td> <a href=" {% url 'delete_task' obj.id %} "> Delete </a>
```

Hemos añadido un hipervínculo que navega a la url que esta llamada como 'delete_task' → delete/<task_id> y el parámetro que le pasamos es obj.id

Editing

urls.py:

```
path('edit/<task_id>', views.edit_task, name='edit_task')
```

En views.py

```
def edit_task(request, task_id):
    if request.method == "POST":
        task = TaskList.objects.get(pk=task_id)
        form = TaskForm(request.POST or None, instance=task)
        if form.is_valid():
            form.save()
            messages.success(request, ("Task edited!"))
            return redirect('todolist')

    else:
        task_obj = TaskList.objects.get(pk=task_id)
        return render(request, 'edit.html', {'task_obj': task_obj})
```

La única diferencia es que hemos creado otra plantilla a la que redirigimos la primera vez que hacemos un GET, una vez allí ya tenemos vinculada la vista, entonces si en esa vista ejecutamos un POST, es decir estamos actualizando una tarea, nos volvemos a redirigir a todolist.

En el HTML hay únicamente un form, pero ¡jojo!, hemos tenido que tener un input escondido, ya que si no al tener el done por defecto a false, al actualizar algo que estaba a true, se ponía a false:

```
{% extends "base.html" %}

{% block title %}
<title>
    Edit Task - Taskmate
```

```

</title>
{% endblock title %}

{% block content %}
    <div class="container">
        <br>
        <form method="POST">
            {% csrf_token %}
            <div class="form-group">
                <input type="text" class="form-control" name=
                <input type="hidden" name="done" value = "{{t
            </div>

            <div class="mt-2">
                <button type="submit" class="btn btn-primary">
            </div>
        </form>
    </div>

{% endblock content %}

```

Accounts and Authentication

1. Hemos creado una aplicación nueva que se encargará de gestionar todo el tema de los usuarios.

```
python manage.py startapp users_app
```

Ahora como hicimos antes, tenemos que modificar el `settings.py` y añadir esta nueva aplicación, y ahora simplemente vincular que todas las llamadas a `domain/account` vayan redirigidas a las urls de esta aplicación

```
path('account/', include('users_app.urls')),
```

Registration

Los primeros pasos que hemos seguido han sido, crear una vista para `account/register` y vincular la vista correspondiente a una plantilla `register.html`,

le pasamos como contexto una clase auxiliar de Django, `UserForm()`, que nos ayudará en esta misión.

En `users_app.urls.py`:

```
urlpatterns = [  
    path('register', views.register, name='register'),  
]
```

En `views.py` de `users_app`:

```
from django.shortcuts import render  
from django.contrib.auth.forms import UserCreationForm  
  
# Create your views here.  
  
def register(request):  
    if request.method == 'POST':  
        register_form = UserCreationForm(request.POST)  
        if register_form.is_valid():  
            register_form.save()  
            messages.success(request, ('User account created  
            return redirect('register')  
        else:  
            register_form = UserCreationForm()  
            return render(request, 'register.html', {'register_form':
```

El template `register.html`:

```
{% extends "base.html" %}  
  
{% block title %}  
<title>  
    Register - Taskmate  
</title>  
{% endblock title %}  
  
{% block content %}  
    <div class="container">
```

```

</br>
{% if messages %}

    {% for message in messages %}
        <div class="alert alert-info alert-dismissible fa
            {{message}}
            <button type="button" class="btn-close" data-
        </div>
    {% endfor %}

{% endif %}
<form method="POST" class="mt-5">
    {% csrf_token %}

    {{ register_form.as_p }}
    <button type="submit" class="btn btn-primary">Registe
</form>
</div>

{% endblock content %}

```

Este formulario viene por defecto en django, pero lo podemos modificar para añadir más campos, por ejemplo el email!

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Adding Email Field in Forms

Estabamos usando el `UserCreationForm`, pero vamos a crear uno ahora nosotros, en `users_app.forms.py`:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

class CustomRegisterForm(UserCreationForm):
    email = forms.EmailField(required=True)

    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

Ahora simplemente usamos ese Form.

Mejorando el diseño con crispy forms.

1. Instalamos el paquete

```
pip install crispy-bootstrap5
```

2. Añadimos a las apps instaladas y a la configuración en `settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'todolist_app', # Add the app to the installed apps
    'users_app', # Add the app to the installed apps
    'crispy_forms',
    'crispy_bootstrap5',
]

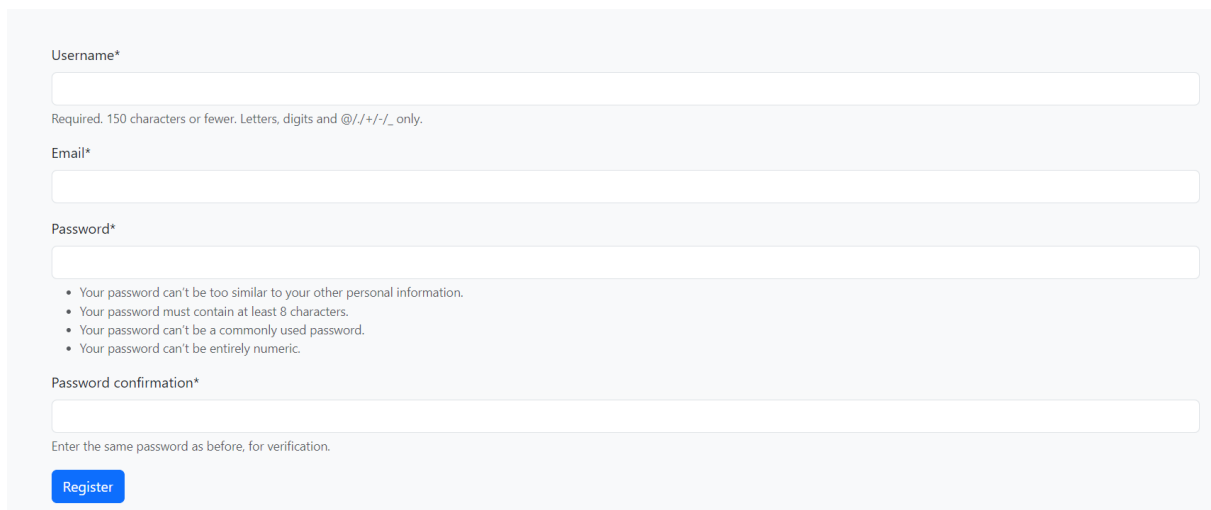
# CRISPY FORMS SETTINGS
```

```
CRISPY_TEMPLATE_PACK = 'bootstrap5'
CRISPY_ALLOWED_TEMPLATE_PACKS = 'bootstrap5'
```

3. Una vez cargado todo, en el template hacemos uso del crispy form añadiendo estas dos líneas, una de load y la otra al renderizar el form:

```
{% load crispy_forms_tags %}

{{ register_form|crispy }}
```



Username*

Required: 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email*

Password*

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

[Register](#)

Login

Para el Login, django nos proporciona ya una vista por defecto!

```
from django.contrib.auth import views as auth_views

path('login', auth_views.LoginView.as_view(template_name='log
```

Simplemente le añadimos el template al que lo queremos vincular y punto:

```
{% extends "base.html" %}

{% load crispy_forms_tags %}

{% block title %}
```



```

<title>
    Login - Taskmate
</title>
{% endblock title %}

{% block content %}
    <div class="container">
        <br>
        <h2 class="mt-3 mb-3">Login On TaskMate</h2>
        {% if messages %}

            {% for message in messages %}
                <div class="alert alert-info alert-dismissible fa
                    {{message}}
                    <button type="button" class="btn-close" data-
                </div>
            {% endfor %}

        {% endif %}
        <form method="POST" class=" col-6">
            {% csrf_token %}

            {{ form|crispy}}
            <button type="submit" class="btn btn-primary mb-5">Lo
        </form>
    </div>

{% endblock content %}

```

Ahora lo que pasamos como contexto se llama 'form'

LOGGING REQUIRED DECORATORS

Son anotaciones que pones en las vistas para limitar quién accede a qué:

```

from django.contrib.auth.decorators import login_required

# Create your views here.

```

```
@login_required
def todolist(request):
```

Ahora tienes que en el `settings.py` decir a donde rediriges si no estas logueado:

```
LOGIN_URL = 'login' # Redirect to the login page if the user
```

RELATIONSHIPS between ENTITIES (models)

```
from django.db import models
from django.contrib.auth.models import User

# Create your models here.

class TaskList(models.Model):
    manage = models.ForeignKey(User, on_delete=models.CASCADE)
    task = models.CharField(max_length=300)
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.task + " - " + str(self.done)
```

Ahora volvemos a hacer migraciones para actualizar la BD, antes borramos todos los datos antiguos, que tienen los datos antiguos sin esta nueva columna.

```
python manage.py makemigrations
python manage.py migrate
```

Como lo vinculo ahora a un usuario en el formulario de nuestra web?

```
@login_required
def todolist(request):

    if request.method == "POST":
        form = TaskForm(request.POST or None)
        if form.is_valid():
```

```

        # El form no tiene el campo manage, lo agregamos
        form.save(commit=False).manage = request.user
        form.save()
        messages.success(request, ("New Task Added!"))
    else:
        messages.success(request, ("The task is too long!"))
        return redirect('todolist')

else:
    all_tasks = TaskList.objects.filter(manage=request.user)
    paginator = Paginator(all_tasks, 6) # Show 6 tasks per page
    page = request.GET.get('pg')
    all_tasks = paginator.get_page(page)
    return render(request, 'todolist.html', {'all_tasks': all_tasks})

```

El formulario TaskForm esta esperando un campo manage, y no lo tiene, lo podemos añadir manualmente con la anotacion commit=False, haciendo que todavia no se guarde en BD, y le agregamos como si fuese un objeto con .manage y le decimos a qué usuario pertenece.

De igual manera ahora mostramos solamente las del usuario de la request:

```
all_tasks = TaskList.objects.filter(manage=request.user)
```

Posibles hackeos 🤖

Primero tenemos que hacer hincapié en que esto se puede gestionar en la vista (lógica de negocio) pero también para aquellas acciones que tienen alguna plantilla intermedia se puede gestionar en el propio HTML que sirve la vista, por ejemplo en el caso edit.

Edit

En el caso Edit, como lo hacemos mediante un formulario, podemos hacer que el HTML que nos envia el servidor sea diferente si somos el usuario o no!

request y task_obj se lo estamos pasando como contexto a este HTML, por tanto podemos acceder a ellos.

```

{% extends "base.html" %}

{% block title %}
<title>
    Edit Task - Taskmate
</title>
{% endblock title %}

{% block content %}
<div class="container">
<br>
    {% if task_obj.manage == request.user %}
    <form method="POST">
        {% csrf_token %}
        <div class="form-group">
            <input type="text" class="form-control" name=
            <input type="hidden" name="done" value = "{{t
        </div>

        <div class="mt-2">
            <button type="submit" class="btn btn-primary"
        </div>
    </form>
    {% else %}
        <h1> You are not allowed to edit this task </h1>
    {% endif %}
</div>

{% endblock content %}

```

Delete, complete and pending

Como no tienen template asociado (podríamos ponerle un PANIC supongo) pero por facilitarnos la vida simplemente vamos a añadir un if a cada vista en la que comprobamos si somos el usuario al que pertenece la tarea que estamos editando, y si lo somos, hacemos la acción y si no pues lanzamos un mensaje de error que nuestra plantilla a la que redirigimos muestra tan normal:

```
task = TaskList.objects.get(pk=task_id)
    if task.manage == request.user:
```

Simplemente con esto ya estamos comprobándolo dentro de cada view.

DATABASE postgres

(Primero borramos el sqlite antes de hacer la 'migración')

DEPLOYMENT guide

Django REST FRAMEWORK