**Development flow:**

* Command to create the project:
  + **django-admin startproject** licitaciones\_web: This starts the project
* Decide what apps do we need:
  + Inicio (landing page, login or sign up, etc.)
  + Proyecto (service or product)
  + User (client or provider)
  + Dashboard (view according to the user role)
* python **manage.py** **startapp** app\_name

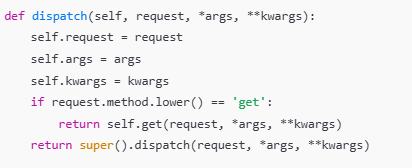
**Django’s built-in views**:

* For the **class-based view “LoginView”:**
  + We do not exactly set up the view in the “views.py” file because is a built-in view, but just put it directly in the “urls.py” of the “home” app. Now this view renders the template on its own context, it only expects Django login form.
    - from **django.contrib.auth** import **views** as auth\_views
    - Then we use “auth\_views.LoginView.as\_view( template\_name = “home/login.html”) in the “urls.py” file of the app.

**Setting and understanding the authentication system**:

* **Django** comes with a **full authentication framework** that provides:
  + User **registration**
  + User **login/logout**
  + Password hashing and checking
  + **Sessions**
  + Third-party auth extensibility (like google, facebook).
* **Parts of Django’s built-in authentication**:
  + “**django.contrib.auth**”: It is the core authentication framework.
  + “**django.contrib.sessions**”: It is **an app that handles sessions**.
  + **User model**: Comes from “**django.contrib.auth.models import User**”. This is **a built-in model for users**.
  + Now we use **“django.contrib.auth import views as auth\_views**”: The “**auth\_views.LoginView**” is a pre-built view that handle login page + login logic.
    - **This view** what it does is that **shows the login form (AuthenticationForm)**, authenticates the user and creates a session for that user to stay logeed in.
    - **The template by default is “registration/login.html”**
  + **Also “django**.contrib.auth import view as auth\_view”: We use auth\_view.LogoutView.
  + **“django**.**contrib.auth.forms** import **UserCreationForm**”: This is a pre-built form for creating users.
    - This **form automatically handles** taking the username and password input, **checking if the username is unique**, **validating password** strength and saving the user.

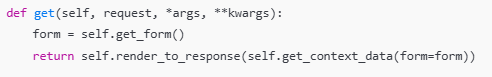
**Generic View in Django**:

* **CreateView**:
  + Basic flow of the CreateView:
    - First, define a model.
    - Then, you can create a form inheriting “forms.ModelForm” or let that part to Django which autogenerates it.
    - Then create a view that inherits “CreateView” and you pass the attributes: “model”, “form\_class” or “fields”, “template\_name” and “success\_url”.
    - Then create the template, do not forget to use “{{% csr\_token %}}
    - “CreateView” is a generic view that inherits from “View”, “TemplateResponsemixin”, “ModelFormMixin” and “BaseCreateView”.
    - **Then under the hood**:
      * First, **Django calls “dispatch()”** which routes the request method that can be “**get()”** or “**post()”.**
        + 
      * **GET METHOD:** 
        + Inside the “**get (request, \*args, \*\*kwargs**)” happens the preparation of the form and renders the form template**.**

We set a **self.object to None** and then return “super().get(request, \*args, \*\*kwargs” that comes from “ModelFormMixin”:



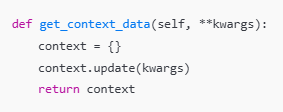
Then the “get()” from the “ModelFormMixin” that sets the **“form” variable** with the **method “get\_form()”** that builds the form instance with empty data.



Then this method “get()” returns render\_to\_response using “**get\_context\_data (form = form)”**.

Texto

El contenido generado por IA puede ser incorrecto.



The “form” object passed as a context variable has “form.fields” that contains a dictionary of fields, “form.data” (POST data if bound), “form.is\_bound”, “form.errors”, “form.cleaned\_data”, “form.instance” (empty in GET and populated in POST) and “form.prefix” and “form.initial” (dict of initial values per field).

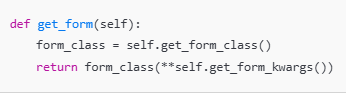
* + - * + The **method “get\_form()”:**

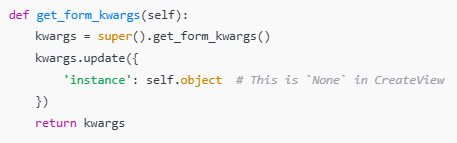
Inside returns the “**form\_class**” or a generated **ModelForm**.

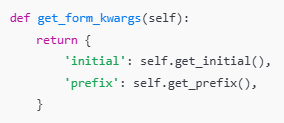
Then return the “form\_class” passing “get\_form\_kwargs()” as an argument.

The function “get\_form\_kwargas()” returns “kwargs”, which is a dictionary.

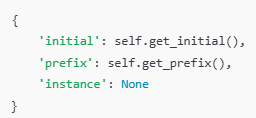
At this point it returns: form\_class(\*\*kwargs).







So, at the end it returns a dict like this:

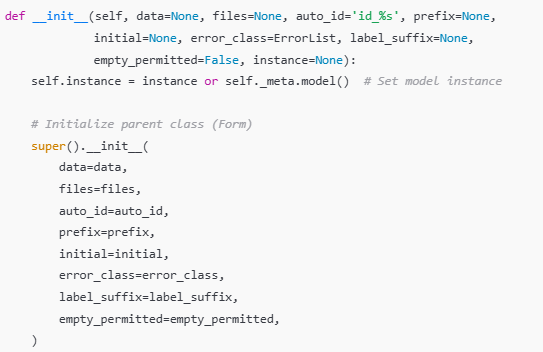


The dictionary has **“initial” key with self.get\_initial()** that **can be used for pre-fill form values**, “prefix”, “data” or “files”.

In a GET request “initial” is {}, “prefix” is None and “instance” is None.

At this point a **QuoteForm is instantiated with**: No POST data, bound= False and initial and prefix dictionary.

When the QuoteForm is instantiated **inside happens “\_\_init\_\_”** that prepares all form fields and attaches this attribute “fields” to the form instance also it creates the attribute “instance” to the form instance.



In this case “instance” is None so it **creates a new blank model object** like this “self.instance = Quote()”.

Inside this also is called **“\_\_init\_\_” from the parent class** that sets “self.data, self.files, etc.” and **builds the “self.fields” dictionary** (from model fields).

**Note**: When the **form class (QuoteForm) is instantiated**, it reads **“Meta.fields” to create “self.base\_fields” that is basically a dictionary.** Then when “\_\_init\_\_()” happens it takes “self.base\_fields” and copies the dictionary to “self.fields”.

* + - * + So, **this can be resume like this**:

**CreateView**: Form\_class\_name()

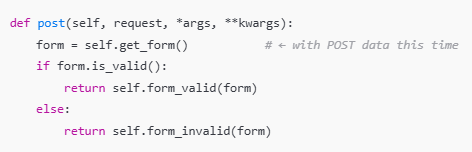
**UpdateView**: Form\_class\_name(**instance = self.object, initial = {…}**)

* + - * + In the template with the context variable “form”:

You cannot access the form fields with “**form.fields[“field\_name”]”** because this is **just the definition of the field** (CharField, DateField, etc.), it **is no connected to data or HTML rendering**.

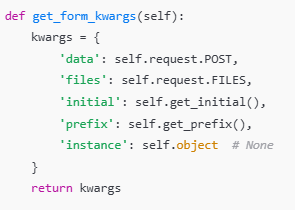
* + - * **POST REQUEST:** 
        + First, **runs “dispatch**” to determine the HTTP method and to choose the correct method.
        + Inside the “post()”:



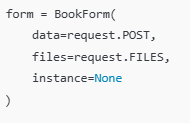


The “get\_form” happens to create the form object, but using the request.POST.

* + - * + Inside the “get\_form” we get the “form\_class” and return the “form\_class” but passing “self.get\_form\_kwargs()”.



Now **the form object results like this:**



* + - * + When the “\_\_init\_\_” method of the Form class happens, the “instance” attribute is attached to the **‘form’ instance**.

In a CreateView this **self.instance = Quote()**.

The attribute “self.fields” is created based on “self.base\_fields”.

Then it associates data with the form fields, but does not populate “instance”.

* + - * + Then “form.is\_valid(form)” runs “form.full\_clean()”:

Here the fields are validated.

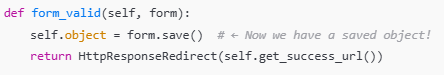
Populated “form.cleaned\_data”.

If valid: cleaned values are passed to “form.\_post\_clean()”.

This is where “self.instance” gets populated.

Adds errors to “form.\_errors” if invalid.

* + - * + After “is\_valid(form)” happens “form\_valid” where the saving object to the DB happens.



In this part we attach a “object” attribute to the view instance.

* + - **Observations**:
      * The “**self.object**” refers to the **model instance being created or updated**.
      * The “self.kwargs” refers to URL kwargs like pk, slug, etc.
  + **Get request**:
    - Django runs “**get\_form\_class()”** to get the auto-generated form.
    - Django runs “**get\_form()”** to create the form instance.
    - Django runs “**get\_context\_data**”, this **passes form into template** context as **“{‘form’: form}”**
  + **Post request**:
    - Django binds data to the form with “**form = Form\_class(request.POST, request.FILES)**”
    - Checks with “**form.is\_valid (form)” which populates form.cleaned\_data dict** and then if it is valid runs “**form\_valid(form)”,** which saves the object and then redirects to “**success\_url**”.
      * Inside “form\_valid(form)” the “**form.save()”** takes the **validated form data** and then **populates the “instance” attribute of the “form” object** and then save the instance to the database and then returns the save object like “object”.
      * **By convention it uses: “self.object = form.save()”,** because later can use the new instance created to get the “pk” or “id” of that instance.
  + **Template “quote\_form.html**”:
    - Django has a built-in method that renders the entire form’s fields wrapped in “<p>” tags:
      * form.as\_p or form.as\_table or form.as\_ul.
      * The other way is to manually render each field with “**form.attribute\_name**”.
    - Here is how we **attach the list with id “contacto\_list”** to the input with “id\_contacto”:
      * This line basically says that for **the input field use for autocompletion the list with ID** **contacto\_list: contactoInput.setAttribute("list", "contacto\_list");**
      * Texto

        El contenido generado por IA puede ser incorrecto.
* **UpdateView:** 
  + **Flow** of the UpdateView **under the hood**;
    - Calls **dispatch**.
    - Here is the **get method:** 
      * Texto

        El contenido generado por IA puede ser incorrecto.
    - Then builds the form using “**get\_form**” that uses the “FormClass” and passes the **argument “instance = self.object”.**
      * When the class is called it runs **“\_\_init\_\_”** that sets “**self.instance = instance**”, then builds the **self.fields and populates this fields** using values from “self.instance”.
    - Then because the fields of “Cliente” are not in the instance we override “get\_initial”:
      * Here is where we add the extra fields to the form’s initial dictionary (used for pre-population).
      * Where “get\_initial” fits:
        + Texto

          El contenido generado por IA puede ser incorrecto.
        + Interfaz de usuario gráfica, Aplicación, Sitio web

          El contenido generado por IA puede ser incorrecto.
        + In summarize: “get\_initial” runs and returns a dictionary and passes this dictionary to the QuoteForm class for the form to be created, then the “\_\_init\_\_” method uses this to populate the fields.

**Django model class (models.Model):**

* The **method “save()”** is used to **define custom behavior** when **an instance of your model is saved to the database**.
* It is clever to override the “save()” in the model itself, because centralizes the behavior:
  + For example, if someone creates a quote instance in the admin panel it will respect the logic in the model “Quote()”, but if the logic is set in the view in this case it will not respect that.
  + The “form.save()” eventually calls the method “save()” from the model.
  + The **default method “save()”**:
    - Handles saving the model instance to the DB.
    - Handles insert (if new object) or update.
    - **Actions**: **Checks whether the object has a primary key (id),** if **not**, it does an **INSERT** (creating a new row), if yes it does an **UPDATE** (modifying an existing row).

**Notes**:

* The **\*\*kwargs** allows a method to **accept arbitrary named arguments as a dictionary**. It sets a dictionary kwargs with all the key-value pairs.
* If I want to **protect a view from not logged-in users**, use **“@login\_required**” with “**django.contrib.auth.decorators**”. If **the user is not logged-in** the user gets **redirect to the “LOGIN\_URL”** set in the settings.py. **This only works with functions not classes**. For classes use “LoginRequiredMixin” and you do not have to override “login\_url” because it searches for “settings.LOGIN\_URL”.
* For the variable “**LOGIN\_URL” use the two slashes**, because in this way it gets treated as **an absolute path**.
  + The use of this **“/login/” basically matters in the “settings.py**” of the main project. It basically says Django start from the main root like “example.com/” and go to “login/”. If you use just “login/” django will use the current page you are on and then add the “login/” so this could fail in case you do not have that root.
  + In the “urls.py” you do not set paths to redirect to, here you set url patterns that Django uses to match incoming urls
* **IMPORTANT ABOUT “User” model:** 
  + When you run “python manage.py **migrate**”, Django **creates database tables** for the **default “auth.User” model** (or the custom one). Then links this table to whatever other table you set to. Then if you decide to extend the “User” model and have a custom model, Django will not migrate from the old user table to the new one. Basically, relationships will break.
  + The **best practice is to set a custom model** from the beginning and **point to this model in “settings.py”** using “**AUTH\_USER\_MODEL**”. Even if you set a custom model from the beginning and then add fields, the old users won’t be transferred to the new table.
* **IMPORTANT ABOUT “MIGRATIONS”:** 
  + You have to make sure that in every app created you must have a **“migrations” folder** and a file **“\_\_init\_\_.py”.**
* **IMPORTANT ABOUT “SELF”** in classes:
  + **“Self”** refers to the **current instance of a class**. When you work with **Django class-based views**, the **instance is crea**ted when you called “**as\_view()”.**
    - Example using “Quote ListView”: When you access “/quotes/” **Django internallly does this:** 
      * view\_instance = **QuoteListView()**
      * response = view\_instance**.dispatch(request)**
        + **During “dispatch”** also does this “**self.request = request**”, so it sets an attribute “request” with the request object.
      * From this point, the **“self” refers to the “QuoteListView” instance**.
* The **“fields” attribute** is only **used by views that render forms**, such as **CreateView and UpdateView**.
* **IMPORTANT**: The classes are resolved left to right.

**Environment config**:

* First install Django with “**pip install django**”. This way we install the latest version.
* Then we create the project with “**django-admin startproject config .”** or “**django-admin startproject name\_project**”. This will define the project structure.
* When we create the app, **do not forget to edit the “urls.py”** and the “settings.py” files to add the apps.
  + Here we ad the **path for “admin/”,** also **path for “accounts/login”**, “accounts/” to include and “**django.contrib.urls**” and “**allauth.urls**”.
    - Here **we override the “login/” url that belongs to “django.contrib.urls” and “allauth.urls”** so that **we can use the “LoginView”** instead of using the “login/” form “allauth.urls”.
  + If we use only “**allauth.urls**” Django expects the templates to be under “**account**”.
  + We add a path: path(**'accounts/3rdparty/login/cancelled/**',**RedirectView.as\_view**(url=reverse\_lazy('login'), permanent=False), name='socialaccount\_login\_cancelled')
    - We stick to name = “socialaccount\_login\_cancelled” because any calls like **{% url ‘socialaccount\_login\_cancelled’ %}**
* In “**settings.py**”
  + Change **the allowed hosts to ‘\*’**
  + Add the apps created to the “**Installed\_APPS**”.
    - “django.**contrib.admin**”: Django’s built-in admin site.
    - “django.**contrib.auth**”: **Core authentication framework** (user model, login/logout, password hashing, decorators like **“@login\_required”.**
    - “django.contrib.contenttypes”: This is a low level app that tracks all of your model classes in the database. Allows you to build generic relationships.
    - “django.**contrib.sessions**”: Enables session support (store data across requests)
    - “django.**contrib.messges**”: A lightweight **“flash messaging” framewok** for one-time notices.
    - “django.**contrib.staticfiles**”: Helps you **manage and serve static assets** during development.
  + If we set in “TEMPLATES” to be “**DIRS: []”** this tells Django “**Do not look in any global folder templates/”** and if you put “**DIRS: [BASE\_DIR / “templates”]** you are telling Django **to look in a global folder** (your\_project/templates)
  + If we set “**APP\_DIRS”: True** tells Django to look inside each installed app’s templates for templates.
  + A recommendation is to use “DIRS: [BASE\_DIR / “templates”] and APP\_DIRS: True
  + In addition, we add “**LOGIN\_URL**”, “**LOGIN\_REDIRECT\_URL**” and “**LOGOUT\_REDIRECT\_URL**” that are basically for Django’s built-in auth views:
    - “**LOGIN\_URL = reverse\_lazy(‘login’**) ”: Used by **@login\_required** to send anonymous users to a login page, also used by “LoginView” when you use redirect\_to\_login()
    - “LOGIN\_REDIRECT\_URL”: After a successful POST to LoginView, Django will send the user here, unless you passed a “next=” parameter in the query string.
  + **SOCIALACCOUNT\_LOGIN\_ON\_GET = True**. If this is set to “True” then a GET request on “accounts/google/login” will inmediately redirect to the OAuth flow.
  + **ACCOUNT\_UNIQUE\_EMAIL = True**. This ensures that allauth will link a social login (like google) to an existing user.
  + **MEDIA\_URL**: Django uses to expose media files.
    - **And when we use “model\_object**.pdf.url” it constructs “MEDIA\_URL” + “quotes\_pdfs/file\_name.pdf”. The quotes\_pdfs is set the attribute “pdf” of the model “Quote”.
  + **MEDIA\_ROOT** #Actual storage location on disk
* Now **let’s create an admin user**:
  + First, **important if you haven’t run “python manage.py makemigrations**” and “**migrate**”, then probably Django hasn’t created yet all the tables from the buil-in apps like (auth, sessions, etc.), so the table “**auth\_user**” does not exist.
  + Then we create the superuser with “**python mange.py createsuperuser**”
    - User: ronald
    - Password: Ronald

**App “users” config**:

* We create this app to **extend the “User” model** and create a custom one.
  + We define a **class “CustomUser”** and add the fields “RUC” and “Company”.
  + We add in “**settings.py”** the “**AUTH\_USER\_MODEL**” and point to this “**app.model\_name**”, also we add the app with “users.apps.UsersConfig”.
  + We also **register the “CustomUser” model in the admin site**. Look the “users.admin.py” for more information.
* We create the **“template” folder** and inside this create “**accounts/login\_social.html**”.

**App “quotes” config**:

* Then create the CRUD **app called “quotes**” with “django-admin startapp quotes”.
* We **modify the “urls.py” of the app to add the classes** from the “views.py”. Also do not forget to add the app to the main “urls.py”.
* We **create the “owner.py”** file to create the views from generic views like “ListView”, “DetailView”, “CreateView”, “UpdateView” and “DeleteView”.
  + class **OwnerView(ListView)** and so on.
* We create the “**model.py**” file:
  + For the “pdfs” we are using “**FileField**” instead “**BinaryField**”:
    - **FileField**: When you **want to store files in the file system** and a keep a reference (path) in the database and when you want easy access to the file via a URL. **File sizes could be large**. This field of the model stores the file in a special object of type “FileField” that has attributes like “.path”, “.url” and “.name”.
    - **BinaryField**: When you want to store the actual file content in the database (as binary data). The files are small and rarely access.
    - **We set “MEDIA\_ROOT” and “MEDIA\_URL” in “settings**.py”:
      * **MEDIA**\_**ROOT** is where uploaded files are physically stored on the disk.
        + If “BASE\_DIR / “media”” then the location will be main\_project/media
      * **MEDIA\_URL is the base URL to access those files in development.**
        + This will be the URL for the access.
    - Then **change the main “urls.py” only in development**, **not in production**:
      * We add this “static (settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)”. This basically tells Django that if a request comes in for “/media/filename.pdf”, serve it from MEDIA\_ROOT folder.
    - In **production you should not serve media files with Django**, use NGINX or Apache, AWS S3, Google Cloud Storage, etc.
  + We create the **“usuario” field**:
    - Here we are creating a **relationship between the “quote” model and the “User” model**.
    - Here we use “**settings.AUTH\_USER\_MODEL**”, because is a way to **reference the User model**.
    - Here we also use “**on\_delete = models.CASCADE**”, this means that if the user is deleted then all the quotes related are eliminated too.
  + We create a **new model “Cliente”** so we can store “contacto”, “email\_contacto” and “cliente\_empresa”:
    - Then in the **“Quote” model we add a field “cliente” to add an FK** to the “Cliente” model.
      * cliente = models.ForeignKey(Cliente, on\_delete=models.SET\_NULL, null=True, blank=True)
* We **modify the “views.py”:**
  + First with “**QuoteListView**”:
    - We are using “**OwnerListView**” which overrides the “get\_queryset()” method of “**ListView**”.
      * “**ListView**” by **default,** **queries the model**, then creates a **context** with a default variable name “**model\_list**” and then renders a template with a default name “**app/model\_list.html**”.
      * By **default**, **the template** “app/model\_list.html” expects a **context variable** called “**model\_list**”.
      * ‘**ListView**’ by default does not protect from users that are not logged in.
      * “**ListView**” uses “**get\_queryset()**” **to get that data** it will show in the template, that is why we override this in “OwnerListView”.
      * So, in summarize: We use “ListView” as a base layer, then in “OwnerListView” we override the “get\_queryset” method, which is the method that gets the data to be display in the template and finally in the “QuoteListView” we set the model to “Quote” so that it uses only that model.
  + Now with “**QuoteCreateView**”:
    - In “**QuoteCreateView**”:
      * With the “**model= Quote**” we tell Django to elaborate the form based on this model, because it uses “ModelForm” and **internally does this** “**form = ModelForm(model = Quote, fields = …)”.**
      * We define the “**form\_class” or the “fields = [..]”** tells Django which fields of the model to include in the form shown to the user.
      * The **line “template\_name= …”** tells Django which **template to use for both “get” and “post”.** Django **renders this template** with **“form” object** and “**view**”.
      * We override “**get\_context\_data**”:
        + First, we call the “get\_context\_data” to create the context variable ( { ‘form’: form }.
        + Then we set the variable “contactos” with all the objects of “Cliente” model. This is a queryset, basically a list of dictionaries.
        + Then with “list(contactos)” we flat this into one line of a dictionary.
        + Then with “json.dumps()” we convert the python list into a JSON string. Which is basically like this ‘[{“key1”:value1,…},{}]’
      * We override “**form\_save(form)**”:
        + First, we **extract validated data from the form fields**:

Variable1 = **form.cleaned\_data**[‘field\_name’]

* + - * + Then we check in the DB if the “cliente” instance already exists.

If it finds a row with a match for “contacto” then updates the values for “email\_contacto” and “cliente\_empresa”.

Then stores the instance in “cliente” and created is a Boolean if a new instance was created or not.

* + - * + Then with “**form.instance.cliente**” **= cliente**.

The form.instance is the Quote model object.

The **“ModelForm” builds the object “instance” before saving it with “form.save()”.** **The “object” is after “instance”**.

The **“form.instance” always exists**, it **represents the model object that the form is working on**.

* + - * + Then we called the **parent method “form\_valid”** that basically **saves the quote instance in the DB and returns the redirect response**.
* We **create a “forms.py**”:
  + We create the same fields that are in the “Cliente” model so we can show it on the form and also modify the constructor of the form “\_\_init\_\_()”
    - The constructor calls every time a form instantiated: form = QuoteForm().
    - The \*args and \*\*kwargs are passed along to the parent “ModelForm”, this may include “data, instance and initial”.
    - This is what happens in the line “**super().\_\_init\_\_(args, kwargs)**”:
      * **Django builds** a copy of the base\_fields **into “self.fields”.**
      * Then if you passed “request.POST” or “request.FILES”, the data is bound to those fields.
      * Then “self.fields” is available for you to modify it.
      * The forms are defined as unbound (GET) or bound (POST).
    - We can also retrieve the fields “objects” that are inside a dictionary called “fields”. **When the form is instantiated, Django builds the dictionary like this (this is related with the fields list defined inside “Meta”)**:
      * Texto

        El contenido generado por IA puede ser incorrecto.
    - We also override the default widget of the “contacto” field with “form.TextInput” and inside define attributes. So, the HTML looks like this:
      * Interfaz de usuario gráfica

        El contenido generado por IA puede ser incorrecto.

**App “Home”** config:

* First, **we create the “urls.py”** and in this file import “**django.urls** import **path** and also import the views.
  + We add a path to the “HomeView” in the views.
  + Here we add paths for “logout/” and “redirect/”.
* Then we create the **template folder and then “home” folder** to add:
  + The “**main.html**” and the “**register.html**”.
* Also, in the main folder of the app “home” we create the file “**context\_processors.py**” and there we add this code (this is not being used, but let’s leave it for now):
  + from **django.conf** import **settings** as **django\_settings**
  + def **settings(request):**
    - return { ‘settings’: django\_settings}
  + Do not forget to add the context processor in the “settings.py” of the main project.
* We **create the “home/forms.py”** to change the form shown in the **“accounts/login/”:**
  + We change the “urls.py” in the main file: path(**'accounts/login/**',**auth\_views.LoginView.as\_view**(template\_name = "accounts/login\_social.html",**authentication\_form=EmailAuthenticationForm**),name = "login"),
  + Check the “forms.py”.
* Then we **set up the “views.py”** that we set up in the “urls.py”:
  + For the **class HomeView** (View)
    - HomeView inherits from “View” and “ LoginRequiredMixin” class, here we override the “get” method.
    - We import “django.conf import settings” to list all the installed apps with settings.INSTALLED\_APPS.
    - We create a dictionary context with two keys: “islocal” and “installed apps”
  + For the **function “register”:**
    - Here we import the “UserCreationForm” with “from **django.contrib.auth.forms** import **UserCreationForm**”
    - We validate the method in the request with “if request.method = “POST”.
    - Then we create the form with “**form** = **UserCreationForm** (request.POST).
    - Then we validate with “**if form.is\_valid() :**” and then we save it with “**form.save()”** this automatically saves it to the database.
  + In addition, we **can add the login option with “Google**”:
    - First, we install “**pip install django-allauth**”, also install **“requests” and “PyJWT”**, because is a dependency of “django-allauth”.
    - Then we update the “settings.py” of the main project:
      * Texto

        El contenido generado por IA puede ser incorrecto.
        + “**sites**” tells Allauth what your current domain is. **Let’s one Django project power multiple domains**. Social login and email confirmations often need to know your current domain name so they can build correct callback URLs.
        + ‘**allauth**’ is the core package for Django All-in-one auth **it wires “local account management and social account workflows”.**
        + ‘**allauth.account’** **handles local accounts** and provides **views, forms and URL**s for “**sign-up, login, logout, e-mail confirmation and password change**”.
        + “**allauth.socialaccount**” is the **social-auth engine**, provides models and views to **store social credentials to Django user accounts**.
        + “**allauth.socialaccount.providers.google**” is the **Google specific provider plugin** (knows how to build Google’s OAuth URLs, fetch the user email, name, etc.). If I want Facebook, Twitter, etc. I would have to install their respective provider packages instead.
      * Also add: **'allauth.account.middleware.AccountMiddleware'** to the **MIDDLEWARE**.
      * Texto

        El contenido generado por IA puede ser incorrecto.
      * Then in **the main “urls.py”** we add various paths #This for Google Login
        + path( ‘**accounts/login’**, **auth\_views.LoginView.as\_view**(template\_name = “account/login\_social.html”), name = “login”) #**This is more specific**.
        + path(**'accounts/**', include(**'django.contrib.auth.urls'**))

Then django.contrib.auth.urls include “accounts/login”, “accounts/logut”, etc.

* + - * + path(**'accounts/',** include('allauth.urls')),

When you use “**allauth.urls”** **Django expects** the templates to be in a **folder called “account/login.html”.**

* + - * + This **set up in the “urls.py”** allows you to **use the template with the built-in login view and the google login button via allauth.**
        + **OBSERVATION:** You might notice that we have put “**accounts/” twice**, but this is not a problem because **Django resolves paths in order and the first match wins**, only if the path inside the included URL matches:

“django.contrib.auth.urls” have “login/”, “logout/”, etc. and “allauth.urls” have “signup/”, “google/login”, etc.

* + - * Then go to “Google Cloud”, create a project and then authorize “Pantalla de consentimiento OAuth”.
      * Client ID: 841639743600-lv75kh0uiu10b4a15hv7nn8p1vpnjdgr.apps.googleusercontent.com
      * Secreto de cliente: GOCSPX-Ys4202\_eVMK-h4lJtv3lE6EKJEGQ
      * Also, under the project just created **add a URL REDIRECT**:
        + http://**127.0.0.1:8000**/accounts/google/login/callback/
      * Then we create an admin account in Django (do not forget to apply makemigrations and migrate).
      * Then we go to “**Social Applications**” and **add “Google”** and here **we add the Client ID and Client Secret**.
      * Also, under “Sites” go to “Sites” and create a new site with a domain name where you have the site.
      * Now, in the “settings.py” set the “SITE\_ID = 2”. Remember that this ID refers to the row number that you have in the “Sites” section in the admin panel and also what you have in the “Social applications” in the configured app the url of the domain site must me in the same position as in “Sites.
      * **Observation:** 
        + If you change your domain “SITE\_ID”, you have to update the site entry in the Django’s admin and your Google console’s redirect URL.
        + When **you enter with “Login with Google”** Allauth **creates a user account automatically** and that user is **associated with the google** account and they do not have a password in the database.

**Template** “**login\_social.html**”:

* Captura de pantalla de un celular

  El contenido generado por IA puede ser incorrecto.
* When you click on “**Login with Google**” **Django-allauth gives you an intermediate “Sign in Via Google” page** so you can confirm or choose an existing account. If you do not want this intermediate page you go to “settings.py” and put “**SOCIALACCOUNT\_LOGIN\_ON\_GET = True**”.
  + By default, **django-allauth’s social login view** **(/accounts/google/login**) behaves like this:
    - **GET request:** Renders an **intermediate form** with the “Continue” button.
    - **POST request**: This actually **redirects out to Google’s OAuth** consent endopoint.
  + When using “**SOCIALACCOUNT\_LOGIN\_ON\_GET = True**” the new url is /accounts/google/login/?process=login and goes directly to Google’s OAuth.
  + When using “{%load socialaccount%}” makes available all the tags defined under the “socialaccount” library including “provider\_login\_url”.
  + The “**provider\_login\_url**” generates the url “accounts/google/login”.

**Template** “**main.html**”:

* Texto

  El contenido generado por IA puede ser incorrecto.
* In this case we are using the **url “account\_logout”** from “**allauth.urls**” in a **POST method**. In this case when I press the “logout” it redirects to “/login/”.

**Deployment flow**

* **Git terms**:
  + **Branch:** It is like **a copy of your project** where you can safely make changes without affecting the original.
    - **The main branch is usually the main version of your code.**
    - You can create a new branch to test a new feature or fix a bug. If works, you merge it back into main.
  + **Local repo**: The **entire project folder plus the git folder**. Is the entire Git structure on your machine, inside this you can have multiple branches.
  + **“.git” folder:**
    - Contains the **commit history**, **branches**, remote connection. It is like Git internal memory and brain.
    - The “**objects/”** is **where the snapshots** (when git commit) of code **live**.
    - Everything Git knows and remembers about your project is stored inside the .git/ folder
  + **Remote repo**: On GitHub. This can also have multiple branches.
  + **Local branch:** This is the branch that exists on your computer. The code and changes you have locally.
  + **Commits:** Every time you save your work in Git; you create a commit. Commits have messages that describe what you changed.
  + **Remote branch:** This is the branch that exists on a remote server like GitHub, GitLab, etc.
  + **Status behind of in front:**
    - Suppose your remote branch is the official version on GitHub.
    - Your local branch can be in different states compared to that remote branch.
    - When the local branch is behind remote branch:
      * Someone else pushed new changes to the remote branch, but you do not have those changes on your local branch yet. You will need to pull those changes to your local branch.
  + **Git commands**:
    - **git init**:
      * Initializes a **new Git repository (The entire project folder plus the .git folder)**.
      * Creates a **hidden “.git” folder**, so Git can **track changes**. Contains all the machinery Git needs. It is like Git’s memory.
    - **git add .:** Stages all your files to be committed. The “.” Means add everything in the current directory. This is like putting all the modified or new files in a box to ship. Basically, you tell Git: “**Prepare these files to be commited**” and **moves the file to the staging area**.
    - **git commit -**m “message”:
      * Saves your staged files into the local repository with a message. **Saves changes locally**.
      * **Git takes a snapshot** of everything in the staging area (git add) and that **snapshot is stored in the local Git repository** (inside the git folder).
    - **git remote add origin https…**: Connects your local Git repo to a GitHub repo. Origin is the name you are giving to this remote (GitHub).
    - **git branch** -M main. This renames the current branch to “main”.
    - **git push** -u origin main: “origin” is the remote GitHub repo, main is the branch name. This line basically uploads your local commits to the “main” branch on Github and sets up a link between your local and remote repo.
  + **How Git saves changes:** 
    - When you commit, Git stores both files as blobs in .git/objects.
    - It stores a tree object representing the folder structure.
    - Then stores the object pointing to that tree.
* First **create a github account and download git** (<https://git-scm.com/download/win>)
* When you install Git:
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
    - This refers to choose the name of the default branch Git will use when you create a new repository with “git init”.
    - We will use “main” instead of “master”.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
    - The **first option** **won’t let me use git commands in the “cmd”** or other tools (like VS Code terminal).
    - The second option lets you use Git from: Git Bash, cmd, PowerShell, VS Code terminal, etc.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
    - Only choose “Use external OpenSSH” if you already have a SSH Client or you want full control over SSH setup.
    - **SSH (Secure shell):** This is a **protocol** used to securely **connect to remote computers and services**, especially for: running commands on a remote server or authenticating with services like GitHub without typing your password every time.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
    - Use the **first option if you are a Windows user**. This keeps your **files compatible with tools on Windows**, but commits clean Unix-style to GitHub.
  + Interfaz de usuario gráfica, Texto, Aplicación

    El contenido generado por IA puede ser incorrecto.
  + Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

    El contenido generado por IA puede ser incorrecto.
* Texto

  El contenido generado por IA puede ser incorrecto.
* Texto

  El contenido generado por IA puede ser incorrecto.
* Then create your repository on GitHub.
* Texto

  El contenido generado por IA puede ser incorrecto.
* In Railway:
  + We create a new project to deploy and choose the github repository.
  + Then we hit “Create” to add postgres DB.
  + Then in the project we create environment variables:
    - DATABASE\_URL
    - SECRET\_KEY
    - EMAIL\_HOST\_USER
    - EMAIL\_HOST\_PASSWORD
  + Then make sure you install “pip install gunicorn”.

We need to modify the settings.py:

* First, we change the DATABASES variable.