# Django

venv\Scripts\activate

django-admin startproject myproject

python manage.py runserver

django-admin startapp app\_name

cd ..

cd file

## Fundamentos:

**MODELS:** a representation of your applications database layout.

All models are subclass of the **django.db.models.Model** class. Each class will be transformed into database tables. Each field is represented by instances of **django.db.models.Field** subclasses (built-in Django core) and will be translated into database columns.

The fields CharField, DateTimeField, etc., are all subclasses of **django.db.models.Field** and they come included in the Django core – ready to be used.

Every model comes with a special attribute, the MODEL MANAGE. You can access it via the Python attribute objects

board = Board**.**objects**.**create(name**=**'Python', description**=**'General discussion about Python.')

Board**.**objects**.**all() : esto retorna una lista QuerySet

django\_board **=** Board**.**objects**.**get(id**=**1) : retorna el objeto q cumple

Fields:

Charfield: we should always define max\_length

DateTimeField: auto\_now\_add setea el tiempo del momento

ForeignKey: crea relaciones entre modelos a nivel de base de datos. Espera un parámetro posicional con la referencia del modelo relacionado.Tiene el parámetro obligatorio, on\_delete, que hay que definir que ocurre cuando el objeto se elimina; se usa harto models.CASCADE. Related\_name parámetro es usado para crear un relación inversa donde las instancias de la otra clase podrán tener una lista de las instancias quienes le pertenecen. La relación inversa se hace automáticamente, pero es bueno poner nombre. Si ponemos un ‘+’ en related\_name, le estamo diciendo a django q no necesitamos este tipo de relación.

Si no epecificamos primary keys/ids, django los hara automáticamente.

| **Operation** | **Code sample** |
| --- | --- |
| Create an object without saving | board = Board() |
| Save an object (create or update) | board.save() |
| Create and save an object in the database | Board.objects.create(name='...', description='...') |
| List all objects | Board.objects.all() |
| Get a single object, identified by a field | Board.objects.get(id=1) |

**Migrating:** decirle a django que cree la base de datos.

python manage.py makemigrations

python manage.py migrate

**Experimenting**

python manage.py shell

exit()

from boards.models import Board

board **=** Board(name**=**'Django', description**=**'This is a board about Django.')

board**.**save()

**for** board **in** boards\_list:

**print**(board**.**description)

Every Django model comes with a special attribute; we call it a **Model Manager**. You can access it via the Python attribute objects. It is used mainly to execute queries in the database. For example, we could use it to directly create a new **Board** object

board **=** Board**.**objects**.**create(name**=**'Python', description**=**'General discussion about Python.')

Board**.**objects**.**all()

django\_board **=** Board**.**objects**.**get(id**=**1)

from django.conf import settings

settings**.**BASE\_DIR

'/Users/vitorfs/Development/myproject'

import os

os**.**path**.**join(settings**.**BASE\_DIR, 'templates')

'/Users/vitorfs/Development/myproject/templates'

**VIEWS:** recive un httpRequest y debe entregar un HttpResponse. Se puede utilizar render para interactuar con el html.

**TEMPLATE:** se encargan de renderizar. Del html.

Hay que crear una carpeta llamada templates/

{% for ... in ... %} and {{ variable }}. They are part of the Django Template Language. The {{variable}} renders the name of the board in the html template, generating a dynamic html document.

Open the **settings.py** inside the **myproject** directory and search for the TEMPLATES variable and set the DIRS key to os.path.join(BASE\_DIR, 'templates'):

TEMPLATES **=** [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [

os**.**path**.**join(BASE\_DIR, 'templates')

],

**TESTING:**

python manage.py test --verbosity**=**2

python manage.py test boards.tests.test\_view\_topic\_posts.TopicPostsTests.test\_status\_code

el flag –verbosity es cuantos mensajes manda

If you are using Django 2.x replace  
from django.core.urlresolvers import reverse  
with  
from django.urls import reverse

If there were an uncaught exception, syntax error, or anything, Django would return a status code **500** instead, which means **Internal Server Error**. Now, imagine our application has 100 views. If we wrote just this simple test for all our views, with just one command, we would be able to test if all views are returning a success code, so the user does not see any error message anywhere. Without automate tests, we would need to check each page, one by one.

url = reverse('home')

        response = self.client.get(url)

        self.assertEquals(response.status\_code, 200)

In the second test, we are making use of the resolve function. Django uses it to match a requested URL with a list of URLs listed in the **urls.py** module. This test will make sure the URL /, which is the root URL, is returning the home view.

view = resolve('/')

        self.assertEquals(view.func, home)

**STATIC FILES SETPUP**

In the project root directory, alongside with the **boards**, **templates**, and **myproject** folders, create a new folder named **static**, and within the **static** folder create another one named **css**:

In your computer, extract the **bootstrap-4.0.0-beta-dist.zip** file you downloaded from the Bootstrap website, copy the file **css/bootstrap.min.css** to our project’s css folder:

The next step is to instruct Django where to find the static files. Open the **settings.py**, scroll to the bottom of the file and just after the STATIC\_URL, add the following:

STATIC\_URL **=** '/static/'

STATICFILES\_DIRS **=** [

os**.**path**.**join(BASE\_DIR, 'static'),

]

**{%** load static **%}<!DOCTYPE html>**

<html>

<head>

<meta charset="utf-8">

<title>Boards</title>

<link rel="stylesheet" href="**{%** static 'css/bootstrap.min.css' **%}**">

</head>

<body>

*<!-- body suppressed for brevity ... -->*

</body>

</html>

First we load the Static Files App template tags by using the {% load static %} in the beginning of the template.

The template tag {% static %} is used to compose the URL where the resource lives

The {% static %} template tag uses the STATIC\_URL configuration in the **settings.py** to compose the final URL. For example, if you hosted your static files in a subdomain like **https://static.example.com/**, we would set the STATIC\_URL=https://static.example.com/ then the {% static 'css/bootstrap.min.css' %} would return <https://static.example.com/css/bootstrap.min.css>.

#### **Introduction to Django Admin**

winpty python manage.py createsuperuser

http://127.0.0.1:8000/admin

# Advanced concepts

#### **URLs**

El orden en urlpatterns importa

When Django receives a request, it starts searching for a match in the project’s URLconf. It starts with the first entry of the urlpatterns variable, and test the requested URL against each url entry.

If Django finds a match, it will pass the request to the **view function**, which is the second parameter of the url. The order in the urlpatterns matters, because Django will stop searching as soon as it finds a match. Now, if Django doesn’t find a match in the URLconf, it will raise a **404** exception, which is the error code for **Page Not Found**.

This is the anatomy of the url function:

**def** **url**(regex, view, kwargs**=**None, name**=**None):

*# ...*

* **regex**: A regular expression for matching URL patterns in strings. Note that these regular expressions do not search **GET** or **POST** parameters. In a request to **http://127.0.0.1:8000/boards/?page=2** only **/boards/** will be processed.
* **view**: A view function used to process the user request for a matched URL. It also accepts the return of the **django.conf.urls.include** function, which is used to reference an external **urls.py** file. You can, for example, use it to define a set of app specific URLs, and include it in the root URLconf using a prefix. We will explore more on this concept later on.
* **kwargs**: Arbitrary keyword arguments that’s passed to the target view. It is normally used to do some simple customization on reusable views. We don’t use it very often.
* **name**: A unique identifier for a given URL. This is a very important feature. Always remember to name your URLs. With this, you can change a specific URL in the whole project by just changing the regex. So it’s important to never hard code URLs in the views or templates, and always refer to the URLs by its name.

## **Path converters**[¶](https://docs.djangoproject.com/en/3.1/topics/http/urls/#path-converters)

The following path converters are available by default:

* **str** - Matches any non-empty string, excluding the path separator, **'/'**. This is the default if a converter isn’t included in the expression.
* **int** - Matches zero or any positive integer. Returns an **int**.
* **slug** - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, **building-your-1st-django-site**.
* **uuid** - Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, **075194d3-6885-417e-a8a8-6c931e272f00**. Returns a [**UUID**](https://docs.python.org/3/library/uuid.html#uuid.UUID) instance.
* **path** - Matches any non-empty string, including the path separator, **'/'**. This allows you to match against a complete URL path rather than a segment of a URL path as with **str**.

TESTING URLS

**class** **BoardTopicsTests**(TestCase):

**def** **setUp**(self):

Board**.**objects**.**create(name**=**'Django', description**=**'Django board.')

**def** **test\_board\_topics\_view\_success\_status\_code**(self):

url **=** reverse('board\_topics', kwargs**=**{'pk': 1})

response **=** self**.**client**.**get(url)

self**.**assertEquals(response**.**status\_code, 200)

**def** **test\_board\_topics\_view\_not\_found\_status\_code**(self):

url **=** reverse('board\_topics', kwargs**=**{'pk': 99})

response **=** self**.**client**.**get(url)

self**.**assertEquals(response**.**status\_code, 404)

**def** **test\_board\_topics\_url\_resolves\_board\_topics\_view**(self):

view **=** resolve('/boards/1/')

self**.**assertEquals(view**.**func, board\_topics)

A few things to note here. This time we used the setUp method. In the setup method, we created a **Board** instance to use in the tests. We have to do that because the Django testing suite doesn’t run your tests against the current database. To run the tests Django creates a new database on the fly, applies all the model migrations, runs the tests, and when done, destroys the testing database.

Para shortcut en error 404 views.py

From django.shortcuts import get\_object\_or\_404

**def** **board\_topics**(request, pk):

board **=** get\_object\_or\_404(Board, pk**=**pk)

**return** render(request, 'topics.html', {'board': board})

class HomeTests(TestCase):

    def setUp(self):

        self.board = Board.objects.create(name='Django', description='Django board.')

        url = reverse('home')

        self.response = self.client.get(url)

    def test\_home\_view\_status\_code(self):

        self.assertEquals(self.response.status\_code, 200)

    def test\_home\_url\_resolves\_home\_view(self):

        view = resolve('/')

        self.assertEquals(view.func, home)

    def test\_home\_view\_contains\_link\_to\_topics\_page(self):

        board\_topics\_url = reverse('board\_topics', kwargs={'pk': self.board.pk})

        self.assertContains(self.response, 'href="{0}"'.format(board\_topics\_url))

#### **Reusable Templates**

This is going to be our master page. Every template we create, is going to **extend** this special template. Observe now we introduced the {% block %} tag. It is used to reserve a space in the template, which a “child” template (which extends the master page) can insert code and HTML within that space.

In the case of the {% block title %} we are also setting a default value, which is “Django Boards.” It will be used if we don’t set a value for the {% block title %} in a child template.

{% load static %}<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8">

    <title>{% block title %}Django Boards{% endblock %}</title>

    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">

  </head>

  <body>

    <div class="container">

      <ol class="breadcrumb my-4">

        {% block breadcrumb %}

        {% endblock %}

      </ol>

      {% block content %}

      {% endblock %}

    </div>

  </body>

</html>

En home.html

**{%** **extends** 'base.html' **%}**

{% block title %}

  {{ board.name }} - {{ block.super }}

{% endblock %}

**{%** **block** breadcrumb **%}**

<li class="breadcrumb-item active">Boards</li>

**{%** **endblock** **%}**

In the **topics.html** template, we are changing the {% block title %} default value. Notice that we can reuse the default value of the block by calling {{ block.super }}. So here we are playing with the website title, which we defined in the **base.html** as “Django Boards.” So for the “Python” board page, the title will be “Python - Django Boards,” for the “Random” board the title will be “Random - Django Boards.”

FORMS

In the <form> tag, we have to define the method attribute. This instructs the browser on how we want to communicate with the server. The HTTP spec defines several request methods (verbs). But for the most part, we will only be using **GET** and **POST** request types.

**GET** is perhaps the most common request type. It’s used to *retrieve* data from the server. Every time you click on a link or type a URL directly into the browser, you are creating a **GET** request.

**POST** is used when we want to change data on the server. So, generally speaking, every time we send data to the server that will result in a change in the state of a resource, we should always send it via **POST** request.

Django protects all **POST** requests using a **CSRF Token** (Cross-Site Request Forgery Token). It’s a security measure to avoid external sites or applications to submit data to our application. Every time the application receives a **POST**, it will first look for the **CSRF Token**. If the request has no token, or the token is invalid, it will discard the posted data.

Here is how we retrieve the data:

subject **=** request**.**POST['subject']

message **=** request**.**POST['message']

The reason why we used board.topics.all instead of just board.topics is because board.topics is a **Related Manager**, which is pretty much similar to a **Model Manager**, usually available on the board.objects property. So, to return all topics associated with a given board, we have to run board.topics.all(). To filter some data, we could do board.topics.filter(subject\_\_contains='Hello').

Another important thing to note is that, inside Python code, we have to use parenthesis: board.topics.all(), because all() is a method. When writing code using the Django Template Language, in an HTML template file, we don’t use parenthesis, so it’s just board.topics.all.

The second thing is that we are making use of a ForeignKey:

**{{** topic.starter.username **}}**

Just create a *path* through the property using dots. We can pretty much access any property of the **User** model. If we wanted the user’s email, we could use topic.starter.email.

**def** **setUp**(self):

Board**.**objects**.**create(name**=**'Django', description**=**'Django board.')

User**.**objects**.**create\_user(username**=**'john', email**=**'john@doe.com', password**=**'123') *# <- included this line here*

*# ...*

**def** **test\_csrf**(self):

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

response **=** self**.**client**.**get(url)

self**.**assertContains(response, 'csrfmiddlewaretoken')

**def** **test\_new\_topic\_valid\_post\_data**(self):

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

data **=** {

'subject': 'Test title',

'message': 'Lorem ipsum dolor sit amet'

}

response **=** self**.**client**.**post(url, data)

self**.**assertTrue(Topic**.**objects**.**exists())

self**.**assertTrue(Post**.**objects**.**exists())

**def** **test\_new\_topic\_invalid\_post\_data**(self):

'''

Invalid post data should not redirect

The expected behavior is to show the form again with validation errors

'''

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

response **=** self**.**client**.**post(url, {})

self**.**assertEquals(response**.**status\_code, 200)

**def** **test\_new\_topic\_invalid\_post\_data\_empty\_fields**(self):

'''

Invalid post data should not redirect

The expected behavior is to show the form again with validation errors

'''

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

data **=** {

'subject': '',

'message': ''

}

response **=** self**.**client**.**post(url, data)

self**.**assertEquals(response**.**status\_code, 200)

self**.**assertFalse(Topic**.**objects**.**exists())

self**.**assertFalse(Post**.**objects**.**exists())

First thing, the **tests.py** file is already starting to get big. We will improve it soon, breaking the tests into several files. But for now, let’s keep working on it.

* **setUp**: included the User.objects.create\_user to create a **User** instance to be used in the tests
* **test\_csrf**: since the **CSRF Token** is a fundamental part of processing **POST** requests, we have to make sure our HTML contains the token.
* **test\_new\_topic\_valid\_post\_data**: sends a valid combination of data and check if the view created a **Topic** instance and a **Post** instance.
* **test\_new\_topic\_invalid\_post\_data**: here we are sending an empty dictionary to check how the application is behaving.
* **test\_new\_topic\_invalid\_post\_data\_empty\_fields**: similar to the previous test, but this time we are sending some data. The application is expected to validate and reject empty subject and message.

**def** **test\_contains\_form**(self): *# <- new test*

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

response **=** self**.**client**.**get(url)

form **=** response**.**context**.**get('form')

self**.**assertIsInstance(form, NewTopicForm)

**def** **test\_new\_topic\_invalid\_post\_data**(self): *# <- updated this one*

'''

Invalid post data should not redirect

The expected behavior is to show the form again with validation errors

'''

url **=** reverse('new\_topic', kwargs**=**{'pk': 1})

response **=** self**.**client**.**post(url, {})

form **=** response**.**context**.**get('form')

self**.**assertEquals(response**.**status\_code, 200)

self**.**assertTrue(form**.**errors)

Now we are using the assertIsInstance method for the first time. Basically we are grabbing the form instance in the context data, and checking if it is a NewTopicForm. In the last test, we added the self.assertTrue(form.errors) to make sure the form is showing errors when the data is invalid.

Creating forms de right waaay

The Forms API is available in the module django.forms. Django works with two types of forms: forms.Form and forms.ModelForm. The Form class is a general purpose form implementation. We can use it to process data that are not directly associated with a model in our application. A ModelForm is a subclass of Form, and it’s associated with a model class.

**class** **NewTopicForm**(forms**.**ModelForm):

message **=** forms**.**CharField(widget**=**forms**.**Textarea(), max\_length**=**4000)

**class** **Meta**:

model **=** Topic

fields **=** ['subject', 'message']

This is our first form. It’s a ModelForm associated with the **Topic** model. The subject in the fields list inside the **Meta** class is referring to the subject field in the **Topic** class. Now observe that we are defining an extra field named message. This refers to the message in the **Post** we want to save.

**def** **new\_topic**(request, pk):

board **=** get\_object\_or\_404(Board, pk**=**pk)

user **=** User**.**objects**.**first() *# TODO: get the currently logged in user*

**if** request**.**method **==** 'POST':

form **=** NewTopicForm(request**.**POST)

**if** form**.**is\_valid():

topic **=** form**.**save(commit**=**False)

topic**.**board **=** board

topic**.**starter **=** user

topic**.**save()

post **=** Post**.**objects**.**create(

message**=**form**.**cleaned\_data**.**get('message'),

topic**=**topic,

created\_by**=**user

)

**return** redirect('board\_topics', pk**=**board**.**pk) *# TODO: redirect to the created topic page*

**else**:

form **=** NewTopicForm()

**return** render(request, 'new\_topic.html', {'board': board, 'form': form})

First we check if the request is a **POST** or a **GET**. If the request came from a **POST**, it means the user is submitting some data to the server. So we instantiate a form instance passing the **POST** data to the form: form = NewTopicForm(request.POST).

Then, we ask Django to verify the data, check if the form is valid if we can save it in the database: if form.is\_valid():. If the form was valid, we proceed to save the data in the database using form.save(). The save() method returns an instance of the Model saved into the database. So, since this is a **Topic** form, it will return the **Topic** that was created: topic = form.save(). After that, the common path is to redirect the user somewhere else, both to avoid the user re-submitting the form by pressing F5 and also to keep the flow of the application.

Now, if the data was invalid, Django will add a list of errors to the form. After that, the view does nothing and returns in the last statement: return render(request, 'new\_topic.html', {'form': form}). That means we have to update the **new\_topic.html** to display errors properly.

If the request was a **GET**, we just initialize a new and empty form using form = NewTopicForm().

The Django Forms API does much more than processing and validating the data. It also generates the HTML for us.

<form method="post">

**{%** csrf\_token **%}**

**{{** form.as\_p **}}**

<button type="submit" class="btn btn-success">Post</button>

</form>

The form have three rendering options: form.as\_table, form.as\_ul, and form.as\_p. It’s a quick way to render all the fields of a form. As the name suggests, the as\_table uses table tags to format the inputs, the as\_ul creates an HTML list of inputs, etc.

Rendering css (bootsrap)

When working with Bootstrap or any other Front-End library, I like to use a Django package called **django-widget-tweaks**. It gives us more control over the rendering process, keeping the defaults and just adding extra customizations on top of it.

Now add it to the INSTALLED\_APPS:

'django.contrib.staticfiles',

'widget\_tweaks',

'boards',

**{%** render\_field field class="form-control" **%}**

The render\_field tag is not part of Django; it lives inside the package we installed. To use it we have to pass a form field instance as the first parameter, and then after we can add arbitrary HTML attributes to complement it. It will be useful because then we can assign classes based on certain conditions.

Now to implement the Bootstrap 4 validation tags, we can change the **new\_topic.html** template:

So, we have three different rendering states:

* **Initial state**: the form has no data (is not bound) Con error o algo mas
* **Invalid**: we add the .is-invalid CSS class and add error messages in an element with a class .invalid-feedback. The form field and the messages are rendered in red.
* **Valid**: we add the .is-valid CSS class so to paint the form field in green, giving feedback to the user that this field is good to go.

##### **Reusable Forms Templates**

In the **templates** folder, create a new folder named **includes**:

Now inside the **includes** folder, create a file named **form.html**:

As the name suggests, the {% include %} is used to *include* HTML templates in another template. It’s a very useful way to reuse HTML components in a project.

The next form we implement, we can simply use {% include 'includes/form.html' %} to render it.

Authentication

The block {% block body %} is wrapping the whole HTML document. We can use it to have an empty document taking advantage of the head of the **base.html**. Notice how we named the end block {% endblock body %}. In cases like this, it’s a good practice to name the closing tag, so it’s easier to identify where it ends.

ime to create the sign up form. Django has a built-in form named **UserCreationForm**. Let’s use it:

{{ field.help\_text|safe }}

Uh, almost there. Currently, our **form.html** partial template is displaying some raw HTML. It’s a security feature. By default Django treats all strings as unsafe, escaping all the special characters that may cause trouble. But in this case, we can trust it.

A basic form processing with a small detail: the **login** function (renamed to **auth\_login** to avoid clashing with the built-in login view).

Everything is working, but… The **email address** field is missing. Well, the **UserCreationForm** does not provide an **email** field. But we can extend it.

LOGOUT

from django.contrib.auth import views as auth\_views

path('signup/', accounts\_views.signup, name='signup'),

We imported the **views** from the Django’s contrib module. We renamed it to **auth\_views** to avoid clashing with the **boards.views**. Notice that this view is a little bit different: LogoutView.as\_view(). It’s a Django’s class-based view. So far we have only implemented views as Python functions. The class-based views provide a more flexible way to extend and reuse views. We will discuss more that subject later on.

Open the **settings.py** file and add the LOGOUT\_REDIRECT\_URL variable to the bottom of the file:

LOGIN

url(r'^login/$', auth\_views**.**LoginView**.**as\_view(template\_name**=**'login.html'), name**=**'login'),

Inside the as\_view() we can pass some extra parameters, so to override the defaults. In this case, we are instructing the **LoginView** to look for a template at **login.html**.

Edit the **settings.py** and add the following configuration:

LOGIN\_REDIRECT\_URL **=** 'home'

##### **Creating Custom Template Tags**

**boards/templatetags/form\_tags.py**

from django import template

register **=** template**.**Library()

@register.filter

**def** **field\_type**(bound\_field):

**return** bound\_field**.**field**.**widget**.**\_\_class\_\_**.**\_\_name\_\_

@register.filter

**def** **input\_class**(bound\_field):

css\_class **=** ''

**if** bound\_field**.**form**.**is\_bound:

**if** bound\_field**.**errors:

css\_class **=** 'is-invalid'

**elif** field\_type(bound\_field) **!=** 'PasswordInput':

css\_class **=** 'is-valid'

**return** 'form-control {}'**.**format(css\_class)

#### **Password Reset**

The idea is during the development of the project, instead of sending real emails, we just log them. There are two options: writing all emails in a text file or simply displaying them in the console. I find the latter option more convenient because we are already using a console to run the development server and the setup is a bit easier.

Edit the **settings.py** module and add the EMAIL\_BACKEND variable to the end of the file:

EMAIL\_BACKEND **=** 'django.core.mail.backends.console.EmailBackend'

The refresh\_from\_db() method make sure we have the latest state of the data. It forces Django to query the database again to update the data. We have to do it because the **change\_password** view update the password in the database. So to test if the password *really* changed, we have to grab the latest data from the database.

#### **Protecting Views**

from django.contrib.auth.decorators import login\_required

@login\_required

**def** **new\_topic**(request, pk):

Notice the query string **?next=/boards/1/new/**. We can improve the log in template to make use of the **next** variable and improve the user experience.

##### **Configuring Login Next Redirect**

**templates/login.html** [(view complete file contents)](https://gist.github.com/vitorfs/1ab597fe18e2dc56028f7aa8c3b588b3#file-login-html-L13)

<form method="post" novalidate>

**{%** csrf\_token **%}**

<input type="hidden" name="next" value="**{{** next **}}**">

**{%** **include** 'includes/form.html' **%}**

<button type="submit" class="btn btn-primary btn-block">Log in</button>

</form>

Then if we try to log in now, the application will direct us back to where we were.

So the **next** parameter is part of a built-in functionality.

We still haven’t really explored Django’s ORM, but the code {{ post.created\_by.posts.count }} is executing a select count in the database. Even though the result is correct, it is a bad approach. Right now it’s causing several unnecessary queries in the database. But hey, don’t worry about that right now. Let’s focus on how we interact with the application. Later on, we are going to improve this code, and how to diagnose heavy queries.

The double underscores topic\_\_board is used to navigate through the models’ relationships. Under the hoods, Django builds the bridge between the Board - Topic - Post, and build a SQL query to retrieve just the posts that belong to a specific board.

Post**.**objects**.**filter(topic\_\_board**=**board)**.**count()

Otro filtro

Post**.**objects**.**filter(topic\_\_board**=**board)**.**order\_by('-created\_at')**.**first()

from django.db.models import Count

from boards.models import Board

board **=** Board**.**objects**.**get(name**=**'Django')

topics **=** board**.**topics**.**order\_by('-last\_updated')**.**annotate(replies**=**Count('posts'))

**for** topic **in** topics:

**print**(topic**.**replies)

2

4

2

1

Here we are using the annotate QuerySet method to generate a new “column” on the fly. This new column, which will be translated into a property, accessible via topic.replies contain the count of posts a given topic has.

\*\*\*\* MAS útil el segundo para contar \*\*\*\*\*

VIEWS CLASS

##### **Class-Based View**

from django.views.generic import View

**class** **NewPostView**(View):

**def** **render**(self, request):

**return** render(request, 'new\_post.html', {'form': self**.**form})

**def** **post**(self, request):

self**.**form **=** PostForm(request**.**POST)

**if** self**.**form**.**is\_valid():

self**.**form**.**save()

**return** redirect('post\_list')

**return** self**.**render(request)

**def** **get**(self, request):

self**.**form **=** PostForm()

**return** self**.**render(request)

urlpatterns **=** [

url(r'^new\_post/$', views**.**NewPostView**.**as\_view(), name**=**'new\_post'),

]

##### **Generic Class-Based View**

CreateView

from django.views.generic import CreateView

**class** **NewPostView**(CreateView):

model **=** Post

form\_class **=** PostForm

success\_url **=** reverse\_lazy('post\_list')

template\_name **=** 'new\_post.html'

url(r'^new\_post/$', views**.**NewPostView**.**as\_view(), name**=**'new\_post')

UpdateView

from django.shortcuts import redirect

from django.views.generic import UpdateView

from django.utils import timezone

@method\_decorator(login\_required, name**=**'dispatch')

**class** **PostUpdateView**(UpdateView):

model **=** Post

fields **=** ('message', )

template\_name **=** 'edit\_post.html'

pk\_url\_kwarg **=** 'post\_pk'

context\_object\_name **=** 'post'

**def** **form\_valid**(self, form):

post **=** form**.**save(commit**=**False)

post**.**updated\_by **=** self**.**request**.**user

post**.**updated\_at **=** timezone**.**now()

post**.**save()

**return** redirect('topic\_posts', pk**=**post**.**topic**.**board**.**pk, topic\_pk**=**post**.**topic**.**pk)

With the **UpdateView** and the **CreateView**, we have the option to either define **form\_class** or the **fields** attribute. In the example above we are using the **fields** attribute to create a model form on-the-fly. Internally, Django will use a model form factory to compose a form of the **Post** model. Since it’s a very simple form with just the **message** field, we can afford to work like this. But for complex form definitions, it’s better to define a model form externally and refer to it here.

The **pk\_url\_kwarg** will be used to identify the name of the keyword argument used to retrieve the **Post** object. It’s the same as we define in the **urls.py**.

If we don’t set the **context\_object\_name** attribute, the **Post** object will be available in the template as “object.” So, here we are using the **context\_object\_name** to rename it to **post** instead. You will see how we are using it in the template below.

In this particular example, we had to override the **form\_valid()** method so as to set some extra fields such as the **updated\_by** and **updated\_at**. You can see what the base **form\_valid()** method looks like here: [UpdateView#form\_valid](https://ccbv.co.uk/projects/Django/1.11/django.views.generic.edit/UpdateView/" \l "form_valid" \t "_blank).

PAGINATOR

from boards.models import Topic

*# All the topics in the app*

Topic**.**objects**.**count()

107

*# Just the topics in the Django board*

Topic**.**objects**.**filter(board\_\_name**=**'Django')**.**count()

104

*# Let's save this queryset into a variable to paginate it*

queryset **=** Topic**.**objects**.**filter(board\_\_name**=**'Django')**.**order\_by('-last\_updated')

from django.core.paginator import Paginator

paginator **=** Paginator(queryset, 20)

*# count the number of elements in the paginator*

paginator**.**count

104

*# total number of pages*

*# 104 elements, paginating 20 per page gives you 6 pages*

*# where the last page will have only 4 elements*

paginator**.**num\_pages

6

*# range of pages that can be used to iterate and create the*

*# links to the pages in the template*

paginator**.**page\_range

range(1, 7)

*# returns a Page instance*

paginator**.**page(2)

**<**Page 2 of 6**>**

page **=** paginator**.**page(2)

type(page)

django**.**core**.**paginator**.**Page

type(paginator)

django**.**core**.**paginator**.**Paginator

paginator**.**page(7)

EmptyPage: That page contains no results

paginator**.**page('abc')

PageNotAnInteger: That page number **is** **not** an integer

page **=** paginator**.**page(1)

*# Check if there is another page after this one*

page**.**has\_next()

True

*# If there is no previous page, that means this one is the first page*

page**.**has\_previous()

False

page**.**has\_other\_pages()

True

page**.**next\_page\_number()

2

*# Take care here, since there is no previous page,*

*# if we call the method `previous\_page\_number() we will get an exception:*

page**.**previous\_page\_number()

EmptyPage: That page number **is** less than 1

##### **GCBV Pagination**

**class** **TopicListView**(ListView):

model **=** Topic

context\_object\_name **=** 'topics'

template\_name **=** 'topics.html'

paginate\_by **=** 20

**def** **get\_context\_data**(self, **\*\***kwargs):

kwargs['board'] **=** self**.**board

**return** super()**.**get\_context\_data(**\*\***kwargs)

**def** **get\_queryset**(self):

self**.**board **=** get\_object\_or\_404(Board, pk**=**self**.**kwargs**.**get('pk'))

queryset **=** self**.**board**.**topics**.**order\_by('-last\_updated')**.**annotate(replies**=**Count('posts') **-** 1)

**return** queryset

While using pagination with class-based views, the way we interact with the paginator in the template is a little bit different. It will make available the following variables in the template: **paginator**, **page\_obj**, **is\_paginated**, **object\_list**, and also a variable with the name we defined in the **context\_object\_name**. In our case this extra variable will be named **topics**, and it will be equivalent to **object\_list**.

Now about the whole **get\_context\_data** thing, well, that’s how we add stuff to the request context when extending a GCBV.

But the main point here is the **paginate\_by** attribute. In some cases, just by adding it will be enough.

TESTS OF CLASS

def test\_board\_topics\_url\_resolves\_board\_topics\_view(self):

        view = resolve('/boards/1/')

        self.assertEquals(view.func.view\_class, board\_topics)

#### **Adding Markdown**

pip install markdown

models

from django.db import models

from django.utils.html import mark\_safe

from markdown import markdown

**class** **Post**(models**.**Model):

*# ...*

**def** **get\_message\_as\_markdown**(self):

**return** mark\_safe(markdown(self**.**message, safe\_mode**=**'escape'))

Here we are dealing with user input, so we must take care. When using the markdown function, we are instructing it to escape the special characters first and then parse the markdown tags. After that, we mark the output string as safe to be used in the template.

##### **Markdown Editor**

<link rel="stylesheet" href="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.css">

<script src="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.js"></script>

**{%** **block** stylesheet **%}**

<link rel="stylesheet" href="**{%** static 'css/simplemde.min.css' **%}**">

**{%** **endblock** **%}**

**{%** **block** javascript **%}**

<script src="**{%** static 'js/simplemde.min.js' **%}**"></script>

<script>

**var** simplemde **=** **new** SimpleMDE();

</script>

**{%** **endblock** **%}**

#### **Humanize**