Desarrollo de la versión inicial

Esta parte mostrará paso a paso el desarrollo del proyecto y su versionado en Git.

1. Instalación

Para usar Django, necesita instalar Python. Este capítulo se escribió utilizando la versión 3.7 del lenguaje. Para Windows y Mac OS, puede encontrar los ejecutables de instalación en la siguiente dirección: https://www.python.org/downloads/release/python-370/

Para distribuciones de Linux basadas en Debian, es posible usar el comando:

```
apt-get install python3.7
```

Para iniciar el intérprete de Python 3.7 en Mac OS y Linux, use el comando:

```
python3
```

En Windows, use el siguiente comando:

```
py
```

Antes de instalar Django, es mejor crear un entorno virtual Python. Este tipo de entorno se utiliza para aislar bibliotecas de Python. Por lo tanto, las bibliotecas instaladas en un entorno virtual no tendrán ningún impacto en el entorno Python del sistema o en otro entorno virtual.

En primer lugar, debe crear la carpeta del proyecto:

```
mkdir simpleblog
```

Luego, debe crear el entorno virtual en la carpeta del proyecto:

```
cd simpleblog
```

Para Linux y Mac OS:

```
python3 -m venv venv_simpleblog
```

Para Windows:

```
py -m venv venv_simpleblog
```

Luego, para utilizar el entorno virtual, es necesario activarlo. En Mac OS y Linux, debe usar el comando:

```
source venv_simpleblog/bin/activate
```

En Windows, debe usar el comando:

```
venv_simpleblog/Scripts/activate
```

Una vez activado el entorno virtual, es posible ejecutar el intérprete de Python 3.7 de la misma forma en Linux, Mac OS y Windows con el comando:

```
python
```

2. Creación del proyecto

El primer paso de desarrollo será la creación del proyecto Django, la inicialización del repositorio Git, la escritura del archivo *.gitignore* adaptado a la tecnología así como la creación de la rama *develop*.

a. Creación del proyecto Django

Para comenzar a desarrollar un proyecto de Django, necesita crear un nuevo proyecto. Para hacer esto, primero debe instalar Django en el entorno virtual. Para ello, debe utilizar el administrador de paquetes Pip (presente por defecto desde Python 3.4):

```
pip install django
```

Para crear el proyecto Django, vaya a la carpeta *simpleblog* y use el siguiente comando:

```
django-admin startproject simpleblog
```

Una vez hecho esto, el comando `ls` debería mostrar las carpetas: *simpleblog* y *venv_simpleblog*.

Este comando creó una carpeta *simpleblog* en la que hay otra carpeta *simpleblog* y un archivo *manage.py*. La carpeta *simpleblog* contendrá elementos específicos del proyecto, como el archivo de configuración (*settings.py*). El archivo *manage.py* es el archivo que se utilizará para manipular Django desde la línea de comandos. Con la ayuda de este archivo es posible:

- iniciar el servidor de desarrollo,
- generar las migraciones,
- ejecutar las migraciones,
- crear un superusuario,
- conectarse a la base de datos,
- etc.

Para el resto de los pasos, debe ir a la carpeta del proyecto e inicializar un repositorio:

```
cd simpleblog
git init
```

b. Creación del archivo .gitignore

En el repositorio recién creado, debe crear el archivo *.gitignore* con el contenido del archivo al que apunta esta URL: https://github.com/github/gitignore/blob/master/Python.gitignore

Este archivo lo proporciona el equipo de GitHub y es el resultado del trabajo de casi 70 colaboradores. Hay muchas otras plantillas de archivo *.gitignore* ofrecidas por GitHub, que se pueden ver en el siguiente repositorio: https://github.com/github/gitignore

Sin embargo, debe permanecer atento: estos archivos corresponden a las necesidades estándar, pero dependiendo de su proyecto y la organización de directorios y archivos, pueden omitir archivos importantes que quiera incluir en la versión. El comando `git status --ignored` le permite mostrar rutas ignoradas por Git (además de la visualización estándar de `git status`). Esto puede ayudar a detectar el uso incorrecto del archivo *.gitignore*.

c. Registro de bibliotecas de Python

La herramienta de gestión de dependencias Pip de Python no proporciona un argumento `--save` (como npm) para guardar las bibliotecas agregadas al proyecto. Sin embargo, es posible mostrar todas las bibliotecas del entorno actual (aquí, el entorno virtual). Para hacer esto, use el siguiente comando:

```
pip freeze
```

Por lo tanto, para guardar las bibliotecas del entorno en el archivo *requirements.txt*, debe usar el siguiente comando:

```
pip freeze > requirements.txt
```

Este archivo tendrá una versión y permitirá que otros desarrolladores potenciales trabajen con las mismas bibliotecas.

d. Primer commit

Después de escribir el archivo *.gitignore*, es hora de crear el primer commit.

Elías primero consulta los archivos que agregará al commit usando el siguiente comando:

```
git status --short --branch
```

Este comando muestra la siguiente salida:

```
## No commits yet on master
?? .gitignore
?? manage.py
?? requirements.txt
?? simpleblog/
```

Esta salida confirma a Elías que los archivos *.gitignore* y *requirements.txt* existen y se agregarán al índice.

No hay un mensaje de commit estándar o perfecto para el primer commit, ya que generalmente no contiene ningún cambio comercial. Cada desarrollador tiene la libertad de definir el mensaje que desee, por fantástico que sea.

```
git add -all
git commit -m "BigBang : simpleblog"
```

Teniendo en cuenta el hecho de que el proyecto es muy simple, Elías decide no utilizar un método Git-Flow completo. Se basará en Git-Flow con las ramas *master* y *develop* usando las ramas *hotfix* y *feature*. No va a implementar ramas de *release*. Por supuesto, si el proyecto se vuelve grande, será posible utilizar futuras ramas *release*.

Por lo tanto, creará un archivo *README* vacío para crear su rama *develop*:

```
git checkout -b develop
touch README
git add README
git commit -m "README: add file"
```

3. Creación de aplicaciones de usuarios y artículos

Las primeras funcionalidades del proyecto dependen mucho unas de otras. Elías usará ramas de *feature* y las fusionará en *develop* antes de crear otra. Esto le permite definir etapas claras en su desarrollo mientras gestiona desarrollos dependientes.

Ahora crea la rama *f-users-articles-app* a partir de *develop*:

```
git checkout -b f-users-articles-app
```

Luego usará el comando `startapp` de Django para crear sus dos aplicaciones (debes haber activado el entorno virtual para hacerlas correctamente):

```
python manage.py startapp users
python manage.py startapp articles
```

Este comando crea dos carpetas en la raíz del repositorio : *users* y *articles*.

Para que las dos aplicaciones se conecten al proyecto principal, es necesario importarlas a la configuración. Para hacer esto, debes editar el archivo *simpleblog/settings.py* y añadir a la lista `INSTALLED_APPS` las aplicaciones:

```
INSTALLED_APPS = [
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles',
  'users',
  'articles',
]
```

Para que las URL de sus aplicaciones sean accesibles desde el proyecto, Elías debe crear el archivo URL de la aplicación del usuario en *users/urls.py* y definir dentro el contenido siguiente:

```
from django.urls import path

app_name = 'users'
urlpatterns = [

]
```

Luego debe definir el archivo URL de la aplicación de artículos creando el archivo *articles/urls.py* con el contenido siguiente:

```
from django.urls import path

app_name = 'articles'
urlpatterns = [

]
```

Luego, Elías editará el archivo *simpleblog/urls.py* para integrar las URL de las aplicaciones en el proyecto definiendo el siguiente contenido:

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

urlpatterns = [
  path(r'^users/', include('users.urls', namespace='users')),
  path(r'^articles/', include('articles.urls', namespace='articles')),
]
```

Ahora que Elías ha creado con éxito las aplicaciones de Django, hace un commit con sus cambios. Para ello, ejecuta los siguientes comandos:

```
git add -all
git commit -m "Apps : add users and articles"
```

Su próximo desarrollo requiere el uso de estas aplicaciones. Por lo tanto, fusionará su rama en *develop* usando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-users-articles-app
```

4. Creación de modelos Django

Uno de los primeros pasos en el desarrollo de un nuevo proyecto es desarrollar modelos de Django. Los modelos serán muy útiles y acelerarán el tiempo de desarrollo aprovechando las características autogeneradas:

- generación automática de migraciones,
- ejecución automática de migraciones en la base de datos,
- generación de la administración.

La gestión de modelos en Django es fundamental. Durante la implementación, esta previsto automatizar la ejecución de migraciones con Git. La presentación de los modelos es sucinta aquí y le permite comprender el marco y la implementación automatizada de Git. Para más información, puede dirigirse a la documentación oficial: https://docs.djangoproject.com/es/2.1/topics/db/models/

Elías llevará a cabo su desarrollo en una nueva rama. Para eso, usa el siguiente comando:

```
git checkout -b f-userprofile-article-models
```

a. El modelo BaseModel

En primer lugar, a veces es útil obtener información sobre los registros de la base de datos. Muchos desarrolladores agregan información a los campos de su base de datos, como la fecha en que se creó el registro o la fecha de la última modificación.

Con Django, es posible crear un modelo abstracto que integre estos comportamientos. Para los modelos en los que es útil tener esta información, será posible definir el modelo abstracto como una clase padre. Para agregar el modelo abstracto, edite el archivo *users/models.py* y agregue la siguiente clase:

```
class BaseModel(models.Model):
  created = models.DateTimeField(
     auto_now_add = True,
     verbose_name="Fecha de creación"
  )
  modified = models.DateTimeField(
     auto_now = True,
     verbose_name="Fecha de modificación"
  )

  class Meta:
     abstract = True
     ordering = ("-created", )
```

Los campos `created` y `modified` contendrán la fecha de creación y la fecha de la última modificación. La clase `Meta` es un elemento específico de Django que le permite agregar comportamientos al modelo. Aquí, el modelo es abstracto (no dará lugar a una tabla en la base de datos) y la ordenación de los datos por defecto se realiza de forma descendente sobre el campo `created` (esta ordenación se puede sobrecargar en los modelos que implementan `BaseModel`).

b. El modelo User

Django ofrece herramientas para administrar fácilmente a los usuarios de un sitio. Aquí, para acelerar el desarrollo, Elías aprovechará al máximo estas herramientas. Debe crear el modelo que gestionará el único usuario destinado al sitio: él mismo. Decide crear su propio modelo de usuario heredando del modelo `AbstractUser` propuesto por Django. En las importaciones del archivo *users/models.py*, debe agregar la siguiente línea:

```
from django.contrib.auth.models import AbstractUser
```

Luego agrega el siguiente modelo al archivo:

```
class UserProfile(AbstractUser, BaseModel):
  displayed_name = models.TextField(
```

```
        verbose_name = "Nombre para mostrar",
        help_text = "Este nombre se mostrará como el autor de los artículos",
        null = True,
        blank = True,
    )

    def __str__(self):
        if self.displayed_name:
            return "%s, %s" % (self.displayed_name, self.username)
        else:
            return "NOT DEFINED, %s" % (self.username)
```

Este modelo utiliza los elementos del modelo `AbstractUser` (documentación: https://docs.djangoproject.com/en/2.1/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser) y agrega la capacidad de elegir un nombre para mostrar. Este modelo hereda de `AbstractUser` y `BaseModel`, lo que significa que el modelo hereda ambos comportamientos. El método `__str__()` genera una cadena de caracteres que representa un registro del modelo. Esto es muy útil en la administración.

Para que este modelo se convierta en el modelo de usuario predeterminado para el proyecto, Elías debe agregar la siguiente línea al archivo *simpleblog/settings.py*:

```
AUTH_USER_MODEL = "users.UserProfile"
```

c. El modelo Article

Este modelo permitirá gestionar los artículos del sitio. Para crear este modelo, Elías agregará la importación de modelos `AbstractUser` y `UserProfile` (así como la importación de `slugify`) al archivo *articles/models.py*:

```
from django.db import models
from django.utils.text import slugify
from users.models import AbstractUser
```

Luego agregará su modelo al archivo:

```
class Article(BaseModel):
  author = models.ForeignKey(
     UserProfile,
     verbose_name = "Autor del artículo",
     on_delete=models.CASCADE,
  )
  title = models.TextField(
     verbose_name = "Titulo del artículo",
     help_text = "Este título se mostrará en la parte superior en h1 y
será el título de la página. Mantenga 50-60 caracteres si es posible.",
     max_length = 100,
  )
  subtitle = models.TextField(
     verbose_name = "Subtítulo del artículo",
     max_length = 100,
  )
  content = models.TextField(
     verbose_name = "Contenido del artículo",
     help_text = "El contenido del artículo esta en HTML",
  )
  slug = models.TextField(
     verbose_name = "URL secundaria del artículo",
     help_text = "Este campo estará presente en la URL del artículo
para ayudar al SEO y dar sentido a la URL.",
  )
  is_display = models.BooleanField(
     verbose_name = "¿Artículo visible?",
     help_text = "Define si el artículo debe poder buscarse",
     default = True,
  )
  is_index = models.BooleanField(
     verbose_name = "¿Artículo indexado?",
     help_text = " Define si el artículo debe indexarse y si se muestra
en la lista de artículos",
     default = True,
  )

  def __str__(self):
     return "%s (%s)" % (self.title, self.author)

  def save(self, *args, **kwargs):
     self.slug = slugify(self.title)
     super(Article, self).save(*args, **kwargs)
```

Este modelo describe el hecho de que un artículo estará compuesto por:

- un autor que corresponde (por una clave externa) a un usuario,
- un título,
- un contenido,
- un campo `slug` que definirá el título que debe aparecer en la URL,
- un campo `is_display` que define si el artículo debe mostrarse,
- un campo `is_index` que define si el artículo debe estar presente en la lista de artículos. Esto le permite administrar artículos para el menú.

El método `save()` permite sobrecargar cuando se guarda el modelo. Esto permite que el campo `slug` se llene desde el campo `Title` eliminando caracteres no autorizados en una URL.

Elías ahora generará las migraciones usando el siguiente comando:

```
python manage.py makemigrations users
python manage.py makemigrations articles
```

Este comando generará los archivos de migración. Para ejecutarlos en la base de datos de desarrollo, Elías utilizará el siguiente comando:

```
python manage.py migrate
```

Ahora que Elías ha definido correctamente sus modelos, hace un commit con sus modificaciones. Después de los controles habituales (`git status`, `git diff`, etc.), ejecuta los comandos siguientes:

```
git add -all
git commit --m "Models : add UserProfile and Article"
```

Su próximo desarrollo requiere el uso de estos modelos. Por lo tanto, fusionará su rama con *develop* utilizando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-userprofile-article-models
```

5. Implementación del módulo de administración

Elías no desea desarrollar una interfaz de administración para su sitio. Prefiere usar la interfaz de administración de Django, que se basa en las plantillas del proyecto y le permite ahorrar mucho tiempo de desarrollo. Esta interfaz se puede configurar fácilmente según los modelos definidos.

Primero comienza creando la rama usando el siguiente comando:

```
git checkout --b f-setup-admin
```

Primero deberá definir la URL que le permitirá acceder a la administración. Para eso, modificará el archivo *simpleblog/urls.py* agregando la siguiente URL:

```
path('admin-blog/', admin.site.urls),
```

Luego modificará el archivo *users/admin.py* con el siguiente contenido:

```
from django.contrib import admin

from .models import UserProfile

admin.site.register(UserProfile)
```

Este archivo se utiliza para agregar una página de administración en la aplicación users relacionada con la administración de usuarios. Por esta parte, Elías sabe que rara vez gestionará usuarios dado que estará solo, por lo que quiere mantener el comportamiento predeterminado de la interfaz sin personalizar nada. Esto significa que la página de administración correspondiente a la gestión de usuarios le permitirá visualizar la lista de usuarios, agregar, modificar y eliminar usuarios. Para la visualización de la lista, es

importante tener bien definido el método `__str__()` del modelo, porque es este método el que se utilizará para visualizar el texto de cada usuario.

Luego, Elías agrega el archivo *articles/admin.py* que le permite definir el comportamiento de las páginas de administración con respecto a la gestión de artículos. Agrega el siguiente contenido al archivo:

```
from django.contrib import admin
from django.contrib.admin import ModelAdmin

from .models import Article

class ArticleAdmin(ModelAdmin):
  search_fields = ('title', )
  list_display = ('author', 'title', 'is_display', 'is_index', )
  list_filter = ('is_display', 'is_index', )
  list_editable = ('is_display', 'is_index', )

admin.site.register(Article, ArticleAdmin)
```

En este archivo, Elías crea una clase que permite sobrecargar el comportamiento predeterminado de la administración. Esta es una de las grandes fortalezas de Django. El framework permite desarrollar comportamientos estándar de manera fácil y rápida. El archivo *users/admin.py* es el ejemplo perfecto, donde una sola línea genera la interfaz de administración de usuarios. Cuando es necesario personalizar los comportamientos predeterminados, Django le permite hacerlo sin tener que codificar todo desde cero. En el archivo *articles/admin.py*, cuyo contenido se muestra arriba, Elías sobrecargó los comportamientos. La interfaz de administración le permitirá buscar un artículo por su título. La página que enumera los artículos mostrará los campos `author`, `title`, `is_display` y `is_index` en forma de tabla. Desde esta página de lista, será posible filtrar las filas según el campo `is_display` y el campo `is_index`. También será posible editar el valor de este campo directamente desde la lista de artículos.

Una vez finalizada la configuración de la interfaz de administración, Elías puede hacer un commit de su código:

```
git add -all
```

```
git commit --m "Users & Articles : setup admin interface"
```

Su próximo desarrollo requiere el uso de este código. Por lo tanto, fusionará su rama en *develop* utilizando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-setup-admin
```

Para poder utilizar el módulo de administración, debe crear un superusuario en la base de datos. Para ello, Django ofrece la siguiente línea de comandos:

```
python manage.py createsuperuser
```

A continuación, se solicita un conjunto de información (nombre de usuario, correo electrónico, etc.). Después de haber validado esta información, es posible conectarse a la interfaz con los identificadores definidos. Para hacer esto, debe iniciar el servidor de desarrollo.

a. Iniciar el servidor de desarrollo

Para probar el proyecto en desarrollo, Django integra un servidor destinado únicamente al desarrollo del proyecto. Este servidor no es adecuado para el uso del proyecto en producción.

Para iniciar este servidor, use el siguiente comando:

```
python manage.py runserver
```

Entonces es posible conectarse al sitio en la URL http://localhost: 8000. En el caso del proyecto de Elías, no existe una URL que apunte a la raíz del proyecto. Para abrir la página de inicio de sesión de administración, debe abrir la URL http://localhost:8000/admin-blog.

Cuando sea necesario que el servidor de desarrollo sea accesible desde la red local, se debe personalizar la dirección IP del comando. El siguiente comando inicia el servidor de

desarrollo en el puerto 8080 y hace que el proyecto sea accesible para la red local:

```
python manage.py runserver 0.0.0.0:8080
```

Es importante tener en cuenta que este servidor no es en absoluto adecuado para el uso en producción. El rendimiento y la seguridad de este servidor solo son adecuados para su uso en desarrollo.

b. Creación de las páginas de usuario

Elías quiere desarrollar un blog muy sencillo. De hecho, la interfaz de usuario solo constará de dos tipos de páginas:

- una página que enumera los artículos con una paginación,
- una página para ver un artículo.

En primer lugar, debe saber que Django usa el modelo MVT (*Model*, *View*, *Template*). Para los desarrolladores acostumbrados a MVC, es fácil de convertir.

Model -> Model

View -> Template

Controller -> View

Entonces, cuando hablamos de vista o view en MVT (Django), corresponde al controlador en el estándar MVC.

c. Templates principales

Antes de desarrollar las páginas del sitio, Elías primero debe crear el template principal. Django ofrece un sistema de templates que le permite crear templates heredados de otros templates tal como las clases de modelos de objetos heredan de otras clases.

Primero comienza creando la rama usando el siguiente comando:

```
git checkout -b f-base-templates
```

Para ello, debe crear el archivo *simpleblog/templates/base.html* con el siguiente contenido:

```
<!DOCTYPE html>
<html lang="es">
 <head>
  <meta charset="utf-8">
  <title>{% block title %}{% endblock title %}</title>
  <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
  <meta name="description" content="">
  <meta name="author" content="">

  {% load staticfiles %}
  <link href="{% static 'lib/css/bootstrap/bootstrap.min.css' %}"
rel="stylesheet">
  <link href="{% static 'lib/css/fontawesome-free/all.min.css' %}"
rel="stylesheet" type="text/css">
  <link href='https://fonts.googleapis.com/css?family=Lora:
400,700,400italic,700italic' rel='stylesheet' type='text/css'>
  <link href='https://fonts.googleapis.com/css?family=
Open+Sans:300italic,400italic,600italic,700italic,800italic,400,
300,600,700,800' rel='stylesheet' type='text/css'>
  <link href="{% static 'lib/css/clean-blog/clean-blog.min.css' %}"
rel="stylesheet">
  {% block head_end %}{% endblock head_end %}
 </head>

 <body>
  <nav class="navbar navbar-expand-lg navbar-light fixed-top"
id="mainNav">
    <div class="container">
     <a class="navbar-brand" href="index.html">Start Bootstrap</a>
     <button class="navbar-toggler navbar-toggler-right"
type="button" data-toggle="collapse" data-target="#navbarResponsive"
aria-controls="navbarResponsive" aria-expanded="false" aria-label=
"Toggle navigation">
```

```
          Menu
          <i class="fas fa-bars"></i>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
          <ul class="navbar-nav ml-auto">
            <li class="nav-item">
              <a class="nav-link" href="">Articulos</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="">Quienes somos </a>
            </li>
          </ul>
        </div>
      </div>
    </nav>

    {% block body_header %}{% endblock body_header %}
    {% block content %}{% endblock content %}
    <hr />
    {% block body_footer %}{% endblock body_footer %}

    <script src="vendor/jquery/jquery.min.js"></script>
    <script src="vendor/bootstrap/js/bootstrap.bundle.min.js"></script>
    <script src="js/clean-blog.min.js"></script>
  </body>
</html>
```

Este archivo *base.html* contendrá toda la estructura mínima del DOM (*Document Object Model*) del sitio. Django permite definir bloques usando la sintaxis `{% block title %}`. Estos bloques serán sobrecargados por los templates que heredarán de él. Los archivos estáticos son administrados por Django usando diferentes sintaxis. Para saber más: https://docs.djangoproject.com/en/2.1/howto/static-files/

Luego creará el archivo *simpleblog/templates/base_blog.html* que contendrá el encabezado del blog. Este encabezado, que contendrá la etiqueta `<h1>`, le permitirá especificar el título del artículo o el título del blog para la página que enumera los artículos. Este archivo contendrá el siguiente contenido:

```
{% extends "base.html" %}
{% block body_header %}
  <header class="masthead" style="background-image: url("{% static
'img/home.jpg' %}")">
    <div class="overlay"></div>
    <div class="container">
     <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
       <div class="site-heading">
        <h1>{% block h1 %}{% endblock h1 %}</h1>
        <span class="subheading">{% block subheading %}{% endblock
subheading %}</span>
       </div>
      </div>
     </div>
    </div>
  </header>
{% endblock body_header %}

{% block body_footer %}
  <footer>
   <div class="container">
    <div class="row">
     <div class="col-lg-8 col-md-10 mx-auto">
      <ul class="list-inline text-center">
       <li class="list-inline-item">
        <a href="#">
         <span class="fa-stack fa-lg">
          <i class="fas fa-circle fa-stack-2x"></i>
          <i class="fab fa-twitter fa-stack-1x fa-inverse"></i>
         </span>
        </a>
       </li>
       <li class="list-inline-item">
        <a href="#">
         <span class="fa-stack fa-lg">
          <i class="fas fa-circle fa-stack-2x"></i>
          <i class="fab fa-linkedin-f fa-stack-1x fa-inverse"></i>
         </span>
        </a>
```

```
</li>
        <li class="list-inline-item">
         <a href="#">
          <span class="fa-stack fa-lg">
           <i class="fas fa-circle fa-stack-2x"></i>
           <i class="fab fa-github fa-stack-1x fa-inverse"></i>
          </span>
         </a>
        </li>
        <li class="list-inline-item">
         <a href="#">
          <span class="fa-stack fa-lg">
           <i class="fas fa-circle fa-stack-2x"></i>
           <i class="fab fa-stack-overflow fa-stack-1x fa-inverse"></i>
          </span>
         </a>
        </li>
       </ul>
       <p class="copyright text-muted">Copyright &copy; BLOG 2018</p>
      </div>
     </div>
    </div>
  </footer>

{% endblock body_footer %}
```

En este archivo, la primera línea se usa para definir que la plantilla hereda de la plantilla *base.html*. La nueva plantilla agrega un bloque de encabezado y un bloque de pie de página. Con estas dos plantillas básicas:

- Si un template hereda de *base_blog.html*, se beneficiará del encabezado y del pie de página.
- Si una plantilla hereda de *base.html*, entonces será libre de no usar un encabezado y pie de página, o definir otros completamente diferentes.

Terminadas los templates básicos, Elías puede hacer un commit con su código:

```
git add -all
git commit -m "Templates : add base and base_blog"
```

Su próximo desarrollo requiere el uso de este código. Por lo tanto, fusionará su rama en *develop* utilizando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-base-templates
```

d. Lista de los artículos

Lo primero que tiene que desarrollar Elías es la vista (por tanto el controlador con el estándar MVC). Para iniciar este desarrollo, Elías primero crea la rama usando el siguiente comando:

```
git checkout -b f-article-list
```

Para hacer esto, debe editar el archivo *articles/views.py* definiendo el siguiente contenido:

```
# coding:utf-8
from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

from .models import Article

def list_articles(request):
  all_articles = Article.objects.filter(is_display=True, is_index=True)
  paginator = Paginator(all_articles, 10)
  page = request.GET.get('page')
  try:
    articles = paginator.get_page(page)
  except PageNotAnInteger:
    articles = paginator.get_page(1)
  except EmptyPage:
    articles = paginator.get_page(paginator.num_pages)
```

```
    return render(
      request,
      'articles/list.html',
      {'articles' : articles, }
      )
```

En esta vista, Elías utiliza el sistema de paginación de Django para simplificar la gestión de la paginación. Esta vista es muy simple:

- recupera los artículos definidos como visibles y correspondientes a la página de paginación,
- gestiona los casos en los que la página enviada no es correcta,
- muestra el template enviándole la lista de artículos usando la función `render()`.

Para poder llamar a esta vista desde la raíz de su sitio, Elías debe hacerla accesible desde su archivo *simpleblog/urls.py* modificando su contenido de la siguiente manera:

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

from articles.views import list_articles

urlpatterns = [
  re_path(r'^articles/', include('articles.urls', namespace='articles')),
  path('', list_articles, name='article:list'),
  re_path(r'^users/', include('users.urls', namespace='users')),
  path('admin-blog/', admin.site.urls),
]
```

Ahora Elías necesita crear su template. Para ello, creará el archivo *simpleblog/templates/articles/list.html* y definir el contenido siguiente:

```
{% extends "base.html" %}
{% block content %}
  <div class="container">
```

```
<div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          {% for article in articles %}
            <div class="post-preview">
              <a href="{% url "articles:detail"
article.slug article.id %}">
                <h2 class="post-title">
                  {{ article.title }}
                </h2>
                <h3 class="post-subtitle">
                  {{ article.subtitle }}
                </h3>
              </a>
              <p class="post-meta">
                Publicado por
                <b>{{ article.author.displayed_name }}</b>
                el {{ article.created }}
              </p>
            </div>
            <hr>
          {% endfor %}
        </div>
      </div>

      <div class="clearfix">
        <span class="step-links">
          {% if articles.has_previous %}
            <a class="btn btn-primary" href="?page=
{{ articles.previous_page_number }}">Precedente</a>
          {% endif %}

          <span class="current">
            Pagina {{ articles.number }} de
{{ articles.paginator.num_pages }}.
          </span>

          {% if articles.has_next %}
            <a class="btn btn-primary" href="?page=
{{ articles.next_page_number }}">Siguente</a>
          {% endif %}
```

```
        </span>
      </div>
    </div>
{% endblock content %}
```

Para probar, después de agregar dos artículos de prueba desde la interfaz de administración, Elías elimina la generación de la URL del template (código `{% url "articles:detail" article.slug article.id %}`) y muestra la página http://localhost:8000. A continuación, vuelve a colocar el enlace para que su código funcione cuando se ha desarrollado la visualización de un artículo.

Una vez completada la funcionalidad para mostrar la lista de elementos, Elías puede hacer un commit con su código:

```
git add -all
git commit -m "Article : list page with pagination"
```

Su próximo desarrollo necesita este código. Por lo tanto, fusionará su rama en *develop* utilizando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-article-list
```

e. Página de consulta de un artículo

Para iniciar la funcionalidad que permite la consulta de un artículo, Elías primero crea la rama usando el siguiente comando:

```
git checkout -b f-article-detail
```

Lo primero que tiene que desarrollar Elías es la vista (por tanto el controlador con el estándar MVC). Para hacer esto, debe editar el archivo *articles/views.py* agregando el siguiente contenido:

```
from django.shortcuts import get_object_or_404

def detail(request, slug, id):
  article = get_object_or_404(Article, id=id, is_display=True)
  return render(
    request,
    'articles/detail.html',
    {'article' : article, }
    )
```

En este archivo, el controlador `detail()` recibe el parámetro `request` (un objeto específico de Django) y un parámetro `pk` (que contendrá el identificador del artículo). Se devuelve un error 404 si no se encuentra el artículo.

Dado que esta vista es específica de la aplicación de artículos, para que sea accesible a través de una URL, Elías debe agregar el siguiente contenido al archivo *articles/urls.py*:

```
from django.conf.urls import include, url

from .views import detail

app_name = 'articles'
urlpatterns = [
  path(r'<slug:slug>-<int:id>', detail, name='detail'),
]
```

Después, Elías tiene que desarrollar su template. Para eso, creará el archivo *simpleblog/articles/detail.html* y definirá el siguiente contenido:

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

{% block h1 %}
  {{ article.title }}
{% endblock h1 %}

{% block subheading %}
  {{ article.subtitle }}
```

```
{% endblock subheading %}

{% block content %}
  <article>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          {{ article.content | safe }}
        </div>
      </div>
    </div>
    <p class="post-meta">
      Publicado por
      <b>{{ article.author.displayed_name }}</b>
      el {{ article.created }}
    </p>
  </article>
{% endblock content %}
```

Elías ha acabado la funcionalidad para mostrar un artículo y ahora puede hacer un commit con su código:

```
git add -all
git commit -m "Article : detail page"
```

Y después puede fusionar su rama en *develop* usando los comandos siguientes:

```
git checkout develop
git merge --no-ff f-article-detail
```

f. Página "Quiénes somos"

A Elías le gustaría agregar una página "Quiénes somos". Esta página le permitiría presentarse y describir los objetivos del blog. No quiere crear una página estática que solo pueda modificar realizando una nueva implementación. Quiere aprovechar al máximo las funciones que ofrece su blog. Por lo tanto, creará el contenido de su página desde su

sistema de blogs y planea que su identificador sea 1.

Para iniciar esta funcionalidad, Elías primero crea la rama usando el siguiente comando:

```
git checkout -b f-article-quienes-somos
```

Modifica su template *simpleblog/templates/base.html* modificando el contenido del enlace de navegación "Quiénes somos":

```
<li class="nav-item">
  <a class="nav-link" href="{% url "articles:detail"
"quienes-somos " 1 %}">Quiénes somos</a>
</li>
```

A partir de ese momento, el primer artículo se convierte en el artículo que se mostrará en el enlace "Quiénes somos".

Terminada la funcionalidad, Elías puede hacer un commit con su código:

```
git add -all
git commit -m "Quienes somos: add nav link"
```

Una vez finalizada la funcionalidad, Elías podrá fusionar su rama en *develop* usando los siguientes comandos:

```
git checkout develop
git merge --no-ff f-article-quienes-somos
```

Para comenzar a funcionar, Elías fusionará la rama *develop* con la rama *master* asignando un tag al commit de fusión:

```
git checkout master
git merge --no-ff develop
git tag v1
```