



Integración de aplicaciones Web basadas en Django.

Django Oscar
+
CMS Wagtail
+
Gadifishing

Cristian Galnares López

Índice

Página

1. Estudio del problema y análisis del sistema	4
1.1. Introducción	4
1.2. Objetivos	4
1.3. Funciones y requisitos	4
1.4. Planteamiento y evaluación de diversas soluciones	4
Saleor Commerce	4
Django Oscar	5
Django Shop 6	
Django CMS	7
Wagatil	8
FeinCMS	9
1.5. Justificación de la solución elegida	9
1.6. Modelado de la solución	10
1.6.1. Recursos humanos	10
1.6.2. Recursos hardware y software	10
1.7. Planificación temporal	11
2. Despliegue del proyecto	11
Instalación de Django Oscar:	12
Creación de un catálogo de productos	13
Instalación de Wagtail CMS	16
Creación del Blog	19
Customización del Blog	21
Integración de Aplicación Web (Gadifishing)	25
3. Fase de pruebas y mejoras	27
Estilo común	27
Barras de navegación (Navbar)	27
Navbar tienda	27
Navbar servicios	28
Footer	29
Importar catalogo	29
Mostrar impuestos	31
Añadir botón de cantidad	32
Pasarelas de pago	34
Integración de PayPal	34
Integración de Stripe	37
Conexión entre Gadifishing y tienda online	40
Cambiar la base de datos	42
Creando una Base de Datos	43
Psycopg2	43
Configuración de Django	43
Otras modificaciones	43

Índice

Página

4. Documentación del sistema	44
4.1 Manual de instalación	45
Añadir HTTPS	45
Agregar Certbot PPA	45
Instalar Certbot	45
Elija cómo desea ejecutar Certbot	46
Renovación automática	46
4.2 Manual de administración	46
Administración del servidor apache	46
Administración de la tienda online	46
Administración Wagtail	48
4.3 Manual de usuario	48
5. Propuestas de mejoras	48
Mejoras en Gadifishing	48
Mejoras en la tienda	48
6. Anexos	49
Anexo 1: Creación de Instancia en AWS	49
Extra: Añadir DNS a nuestra instancia	52
Extra: requisitos para aplicaciones Django	53
Anexo 2: Extendiendo aplicaciones de Django Oscar	54
7. Conclusiones finales	55
8. Bibliografía	55

1. Estudio del problema y análisis del sistema

1.1. Introducción

Este proyecto consiste en la integración de aplicaciones Web basadas en Django. Se creará una coexistencia entre tres aplicaciones; una tienda online que será la base de este proyecto, Gadifishing (aplicación web creada durante el curso, que está destinada a recavar información detallada sobre las capturas en la jornadas de pesca) y la última, un CMS basado en Django para crear y gestionar un blog.

En la tienda online venderemos productos relacionados con la pesca, ademas de ofrecer otros servicios, tales como el registro de la información de nuestras jornadas de pesca y un blog para dar a conocer información relacionada con la tienda o de pesca en general. Con esto, queremos crear una integración de las tres aplicaciones, de forma que se pueda relacionar la información de los productos de la tienda a través de los servicios que ofrece.

Tanto para la integración de la tienda como para el blog, estudiaremos las diversas posibilidades de las que disponemos y realizaremos un estudio detallado de sus pros y sus contras para elegir la aplicación que mas se adapte a nuestras necesidades.

1.2. Objetivos

Con este sistema que vamos a implementar queremos conseguir:

- Dar a conocer nuestros productos y servicios al mayor número de personas para así conseguir mayor venta de nuestros productos.
- Crear una conexión entre todo el sistema de forma que la informacion de todas las aplicaciones estén relacionadas entre sí.
- Un sistema de autenticación unificado, es decir, todas las aplicaciones usarán la misma base de datos de usuario.
- Facilitar a los usuarios un registro con los datos de las capturas realizadas en sus jornadas de pesca.
- Facilitar el acceso y la interacción de nuestro portal con el usuario.

1.3. Funciones y requisitos

- La creación del portal web que integrará todas nuestras aplicaciones y contendrá toda la información de éstas.
- Sistema de administración sencillo de todo el portal (productos,pedidos, envíos, pagos, contenido del blog...)
- Proporcionar un entorno de administración para cada aplicación, de modo que no solo afecte a nivel global, es decir, proporcionar un nivel de privilegios para el encargado de administrar cada aplicación.

1.4. Planteamiento y evaluación de diversas soluciones

Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como MVC (Modelo–Vista–Controlador). Fue desarrollado en origen para gestionar varias páginas orientadas a noticias de la World Company de Lawrence, Kansas, y fue liberada al público bajo una licencia BSD en julio de 2005. En junio de 2008 fue anunciado que la recién formada Django Software Foundation se haría cargo de Django en el futuro.

La meta fundamental de Django es facilitar la creación de sitios web complejos. Django pone énfasis en el re-uso, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio No te repitas (DRY, del inglés *Don't Repeat Yourself*). Python es usado en todas las partes del framework, incluso en configuraciones, archivos, y en los modelos de datos.

Las aplicaciones web son herramientas que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet mediante un navegador. En otras palabras, es un programa que se codifica en un lenguaje interpretable por los navegadores web en la que se confía la ejecución al navegador.

Las aplicaciones web son populares debido a lo práctico del navegador web como cliente ligero, a la independencia del sistema operativo, así como a la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software a miles de usuarios potenciales. Existen aplicaciones como los correos web, wikis, blogs, tiendas en línea... Es importante mencionar que una página web puede contener elementos que permiten una comunicación activa entre el usuario y la información. Esto permite que el usuario acceda a los datos de modo interactivo, gracias a que la página responderá a cada una de sus acciones, como por ejemplo llenar y enviar formularios, participar en juegos diversos y acceder a gestores de base de datos de todo tipo.

En primer lugar veremos las posibles soluciones para integración de la tienda online:

En los propios paquetes¹ que ofrece Django, podemos encontrar numerosas aplicaciones de tienda online. Pasaremos a ver una pequeña descripción de las más utilizadas.

Saleor Commerce



*Saleor*² es una plataforma de comercio electrónico de código abierto de rápido crecimiento que ha prestado servicios a empresas de gran volumen de sucursales como publicaciones y vestimenta desde 2012. Con base en Python y Django, la última actualización importante presenta un front-end modular impulsado por una API GraphQL y escrito con React y TypeScript.

1 Paquetes Django Ecommerce: <https://djangopackages.org/grids/g/ecommerce/>

2 Repositorio Saleor: <https://github.com/mirumee/saleor>

Está compuesta por tres componentes: Saleor platform, storefront y dashboard.

Actualmente tiene una producción estable y es compatible con python3.

Requiere de dos instancias, una para la plataforma y otra para el storefront y dashboard.

Django Oscar



*Django Oscar*³ es un portal de comercio electrónico para Django diseñado para crear sitios basados en dominios. Está estructurado de tal manera que cualquier parte de la funcionalidad principal se puede personalizar para satisfacer las necesidades de su proyecto. Esto permite manejar una amplia gama de requisitos de comercio electrónico, desde sitios B2C a gran escala hasta sitios B2B complejos.

Está compuesta por la plataforma y un dashboard para administrar toda la aplicación.

Dispone de numerosas extensiones creadas y mantenidas por la comunidad, ademas de ser compatible con las pasarelas de pago para Django.

Actualmente tiene una producción estable y es compatible con python3.

Es la aplicación mejor valorada y más utilizada por los usuarios.

Django Shop



*Django shop*⁴es un portal de comercio basado en python y prediseñado para operar junto a Django-CMS. Requiere de una base de datos independiente para su catalogo.

³ Repositorio Django Oscar: <https://github.com/django-oscar/django-oscar>

⁴ Repositorio Django Shop: <https://github.com/awesto/django-shop>

Permite a un comerciante implementar todas las características adicionales deseadas, sin tener que modificar ningún código de django-SHOP. Sin embargo, requiere agregar código personalizado a la implementación comercial.

Actualmente tiene una producción estable y es compatible con python3.

En segundo lugar veremos las posibles soluciones para integración de un blog:

Para la integración de un blog lo haremos a través de un CMS basado en python.

También tenemos la opción de crear el blog directamente en Django pero esto podría ocasionarnos problemas de seguridad, ya que compartiría administración junto al resto del portal.

La creación del blog desde un CMS nos permite una mejor gestión de los contenidos y a su vez podemos separar el panel de administración del blog de los paneles de nuestro portal.

Django CMS



Django CMS⁵ se creó inicialmente por la comunidad de Django y Python para solucionar los problemas de seguridad habituales en los CMS más utilizados y crear una plataforma con un núcleo especialmente ligero. Es además muy fácil de usar, una característica esencial para el usuario final. Los desarrolladores pueden integrar otras aplicaciones desarrolladas con Django con gran facilidad.

Algunas de las características fundamentales de Django CMS:

- Plataforma en varios idiomas por defecto: no es necesario implementar ningún plugin especial para disponer de herramientas, páginas web y contenidos en otros idiomas.
- Interfaz basada en la función de arrastrar y soltar.
- Soporte de pantalla táctil: esto permite que el usuario del CMS pueda crear contenidos desde dispositivos como un móvil o una tableta.
- Interfaz: velocidad mejorada y consumo de ancho de banda bajo.
- Sistema de plugins: cualquier desarrollador puede incorporar nuevas funcionalidades o aplicaciones al CMS. También existe la posibilidad de desarrollar aplicaciones propias e implementarlas dentro del editor.

5 Sitio Web de Django CMS: <https://www.djangoproject-cms.org/en/>

Wagtail



Los creadores de Wagtail⁶ lo definen como un CMS de código abierto en Python, construido en Django “por desarrolladores para desarrolladores”. Debe su nombre a un ave, la motacilla alba o lavandera blanca. Ellos consideran Django la plataforma ideal para el desarrollo de un CMS por su robustez, su facilidad a la hora de picar código y su velocidad en el desarrollo.

Algunas de sus características esenciales:

- Integración de aplicaciones Django dentro del CMS.
- Configuración de los distintos tipos de contenidos a través de los modelos estándar de Django.
- Control sobre el diseño a través de las plantillas de Django.
- Sistema de permisos y roles configurables.
- Plataforma con soporte para varios idiomas.
- Elasticsearch como motor de búsqueda: es un sistema de código abierto que se integra a la perfección en el editor. La interfaz de Wagtail dispone también de un sistema de recopilación de estadísticas a partir de las búsquedas en ese motor. Con él se puede acceder de forma rápida a documentos, páginas, vídeo e imágenes.
- *Streamfield*: sistema de creación de información totalmente libre. Es posible ordenar como se quiera todos los elementos de un contenido: titular, texto en párrafos, imágenes, vídeos, ladellos...

⁶ Sitio Web de Wagtail CMS: <https://wagtail.io/>



FeinCMS⁷ es otro ejemplo de CMS basado en Django. El editor se fundamenta en la capacidad para añadir distintos tipos de contenido a las páginas: contenido en bruto en HTML o JavaScript; contenido en formato vídeo de YouTube o Vimeo (con solo cortar y pegar el enlace se embebe dentro de la página); archivos de audio; listado de comentarios; fuentes RSS...

FeinCMS dispone de una serie de extensiones que añaden funcionalidades:

- Mejorar el posicionamiento en buscadores de contenido (SEO): capacidad para añadir los metadatos de una noticia (title, metadescripción, metaetiquetas...). En CMS más conocidos como Wordpress existen plugins específicos para esta labor.
- Posibilidad de alojar páginas web en varios idiomas: se puede añadir un campo de lenguaje para cada una de las páginas del site.
- Este CMS comparte el panel de administración junto al panel de Django

1.5. Justificación de la solución elegida.

Para justificar las soluciones basándonos en nuestras necesidades, he desplegado y testeado todas las aplicaciones del apartado anterior llegando a las siguientes conclusiones:

Elijo la utilización del lenguaje de programación Python porque es el lenguaje que hemos estudiado con mas profundidad a lo largo de nuestro Ciclo. A su vez elegimos Django como framework para el proyecto por la misma razón.

He decidido utilizar Django Oscar como tienda online y como base de este sistema porque encaja perfectamente en nuestras necesidades de venta y administración. Esta aplicación tiene un despliegue simple y rápido, su código fuente es totalmente configurable y adaptable a cualquier tipo de producto, ofrece un amplio panel de administración para gestionar toda la tienda, numerosas extensiones adaptables... Lo más importante es que mantiene una producción estable, disponen de una gran comunidad y es utilizado por un gran número de desarrolladores.

Por otro lado, decido no utilizar Saleor y Django Shop porque no son tan customizables como Django Oscar y requieren de instancias independientes para su funcionamiento (lo cuál implica un aumento de recursos y rendimiento).

⁷ Repositorio de Fein CMS: <https://github.com/feincms/feincms>

Como CMS basado en Django he decidido utilizar Wagtail, principalmente por hacer una separación en los paneles de administración. Este CMS tiene su propio panel de administración y gestión del blog, no lo comparte con el propio de Django.

Esto es exactamente lo que buscábamos satisfacer, disponer de permisos únicos sobre el blog y sus contenidos, pudiendo separar así la administración de blog a la tienda o al propio portal.

Otra razón es ser 100% customizable y compatible con cualquier aplicación basada en Django, ademas de su fácil instalación.

1.6. Modelado de la solución

En los siguientes apartados veremos como hemos ido adaptando todo el sistema a nuestras necesidades.

1.6.1. Recursos humanos

- Una persona para implementar el portal en un servicio de Web Hosting, que en este caso será una instancia de Amazon Web Server. Esta instancia contará con un entorno virtual donde serán instalados todos los paquetes y dependencias del portal, un servidor apache y un DNS o dominio.
- Una persona dedicada al mantenimiento de AWS.
- Las personas encargadas de la administración de la tienda y los servicios. En este caso, podría ser la misma persona la encargada de toda la administración o bien, pueden ser diferentes personas. Aquí debo resaltar la importancia de la separación de los paneles de administración, por ejemplo, si hay una única persona encargada del blog, solo deberá tener acceso al panel del blog y no al resto de paneles.

1.6.2. Recursos hardware y software

Instancia de AWS⁸ con 4BG de RAM y Ubuntu 18.04

Esta instancia contará con el siguiente software:

Servidor Apache

Software de entornos virtuales (virtualenv)

Paquete npm

Paquete node.js

* *El resto de paquetes y dependencias serán instaladas en el entorno virtual tras el despliegue de la aplicación.*

⁸ Para aprender como crear una instancia en AWS ir al ANEXO 1

1.7. Planificación temporal

Semana del 13 de Abril: investigación de las distintas aplicaciones web disponibles y pruebas de las mismas.

Semana del 21 de Abril a 3 de Mayo: estudio del funcionamiento las opciones elegidas y observar sus posibles modificaciones.

Semana del 3 de Mayo a 10 de Mayo: integración de todas las aplicaciones en un mismo portal.

Semana del 10 de Mayo a 18 de Mayo: creación del catalogo base de la tienda (categorías, tipos de productos, atributos de productos...) y creación del blog.

Semana del 18 de Mayo a 25 de Mayo: mejoras visuales y funcionalidades de la tienda, creación de la instancia de AWS y primeras pruebas de hosting.

Semana del 25 de Mayo a 2 de Junio: implementación de pasarelas de pago y pruebas de funcionamiento.

Semana del 2 de Junio a 9 de Junio ultimas mejoras visuales, interconexion de todos los servicios y añadir funcionalidades.

Semana del 9 de Junio a 16 de Junio puesta en marcha en AWS.

2. Despliegue del proyecto

Para comenzar el despliegue, lo primero que necesitamos es instalar algunos paquetes previos:

Python 3.X

CLI> sudo apt-get install python3.X

Django

CLI> python -m pip install Django

Pip

CLI> apt-get install python3-pip

Npm

CLI> pip install npm

Node.js

CLI> pip install node.js

Apache2

CLI> apt-get install apache2

#Git

CLI> apt-get install git

Se recomienda utilizar entornos virtuales para instalar los paquetes y las dependencias. Esto nos permite una gran número de posibilidades, que van desde realizar test, probar la integración de un módulo con distintas versiones ... hasta realizar despliegues web.

CLI> sudo pip3 install virtualenv

Una vez instalados los paquetes anteriores, podremos comenzar a crear nuestro proyecto en Django. Para ello lo que tenemos que hacer es ejecutar el fichero django-admin.py de la siguiente manera:

CLI> django-admin.py startproject miproyecto

Instalación de Django Oscar:

Para el despliegue de Django Oscar, lo primero será crear un entorno virtual y luego clonar su repositorio.

Para la creación del entorno escribimos:

CLI> virtualenv nombre_de_tu_entorno -p python3

Y ahora lo activamos

CLI> source ./nombre_de_tu_entorno/bin/activate

Ahora pasamos a descargar el repositorio de Django Oscar:

CLI> git clone https://github.com/django-oscar/django-oscar

Nos posicionamos dentro de la carpeta descargada y escribimos “*make sandbox*”.

Con este comando se desencadena el despliegue e instalación de la tienda:

- Dependencias y requisitos
- Nodos js
- Migraciones iniciales
- Ficheros estáticos y plantillas base
- Entornos de testeado

Una vez terminado el despliegue, crearemos un usuario administrador para todo el proyecto:

CLI> python manage.py createsuperuser

Y luego ejecutamos nuestro servidor Web con el comando:

CLI> python manage.py runserver

Ahora ya tenemos nuestro servidor corriendo, para poder verlo podemos visitar con nuestro navegador nuestra dirección ip, o poner “localhost:8000”, o “127.0.0.1:8000”.

Creación de un catálogo de productos

Para poder crear nuestro catálogo⁹ de productos, hablaré sobre como Django Oscar interpreta los productos y sus propiedades.

Productos: para Django Oscar un producto es cualquier artículo que vendemos. Estos productos contarán con atributos comunes a todos ellos (nombre, código, imagen, descripción, stock...)

Categorías: es la forma que tiene de estructurar o catalogar los distintos productos mediante categorías padres y categorías hijas. Por ejemplo la categoría (padre) anzuelos, contará con diversas categorías hijas.

Categoría padre

5 Categorías		
Nombre	Descripción	Número de categorías hijas
Anzuelos		6

Categorías hijas

6 Categorías		
Nombre	Descripción	Número de categorías hijas
Acero inox		0
Aparejos		0
Carbono		0
Montados		0
Triplex		0
Triplex carbono		0

Estas categorías hijas serán las que contengan los productos en cada una de ella.

Tipo de producto: es común que una tienda no se dedique a la venta de un único producto, es decir, venderá distintos tipos (ropa, complementos, zapatos...). Cada tipo de producto cuenta con unas propiedades o atributos concretos (talla, color, tamaño, precio...) que no comparte con otro tipo de producto. Django Oscar nos permite asignar atributos específicos a cada producto mediante los tipos de productos, es decir, todos los productos pertenecientes a un mismo tipo, contarán con los mismos atributos.

Tipo de productos		
Nombre	Requiere envío	Seguir stock?
Anzuelos	sí	sí

9 Ficheros del cataálogo:

https://github.com/imspaiK/documentacion_proyecto/tree/master/proyecto_aplicaciones_web/Catalogo

Y sus atributos comunes al mismo tipo

Actualizar tipo de producto 'Anzuelos'

Atributos de producto		
Nombre	Código	Tipo
Marca	marcas	Opcion Marca Selecione el grupo de opciones
Número	numero	Texto
Unidades por paquete	unidades	Entero

En caso de otro tipo de producto, encontramos atributos diferentes:

Actualizar tipo de producto 'Cañas'

Atributos de producto		
Nombre	Código	Tipo
Acción (lbs)	accion	Número en coma flot...
Largo (m)	largo	Número en coma flot...
Marca	marcas	Opcion Marca Selecione el grupo de opciones
Peso (g)	peso	Número en coma flot...

Opciones de producto: una opción de producto es una opción para el comprador que hace que su precio no cambie. Por ejemplo para el color, talla...

Variantes de producto: una variante podríamos definirlo como un producto que cuenta con un stock y precio distinto en función de una propiedad común. Por ejemplo un zapato de niño y otro de adulto, su precio cambia por el uso del material.

Ahora que hemos conocido un poco como funciona el catálogo, pasamos a crearlo. Para la creación del catálogo, Django Oscar nos permite importarlo mediante ficheros .csv y .json, pero también podemos crearlo desde el panel de administración.

Mediante .csv

Para importar el “fichero.csv” usamos el comando:

```
CLI> python manage.py oscar_import_catalogue fichero.csv
```

Si creamos nuestro catálogo desde este tipo de fichero, importamos los productos a la vez que creamos las categorías y algunos atributos comunes.

```
1 |Tipo de producto, Categoria P > Categoria H, Código, Nombre, Descripcion, Proveedor, Imagen, Precio, Stock  
2 |Anzuelos,Anzuelos > Acero inox,018800,880-SS,"Anzuelo especial Big-Game. Extrafuerte, forjado, recto.",Proveedor,018800,4.99,20  
3 |Anzuelos,Anzuelos > Acero inox,011880,AJI TYPE DOUBLE,"Anzuelo extrafuerte forjado",Proveedor,011880,4.99,20
```

En caso de no existir tipo de producto o categoría, se creará y si existe, se añade el producto al tipo de producto y categoría existente.

Mediante .json

Para importar el fichero.json usamos el comando:

```
CLI> python manage.py manage.py loaddata fichero.json
```

Este método es mas potente que el anterior, ya que nos permite añadir cualquier tipo de objeto de acuerdo a sus atributos en la base de datos. Por ejemplo crear atributos concretos de un tipo de producto, variantes, ofertas, añadir valores...

Ej. Crear superuser

```
{
    "pk": 1,
    "model": "auth.user",
    "fields": {
        "username": "superuser",
        "first_name": "",
        "last_name": "",
        "is_active": true,
        "is_superuser": true,
        "is_staff": true,
        "last_login": "2012-09-12T17:13:49Z",
        "groups": [],
        "user_permissions": [],
        "password": "pbkdf2_sha256$10000$ZMgucl",
        "email": "superuser@example.com",
        "date_joined": "2012-09-12T17:00:02Z"
    }
},
```

Ej. Crear un tipo de producto

```
{
    "pk": 1,
    "model": "catalogue.productclass",
    "fields": {
        "track_stock": true,
        "options": [],
        "requires_shipping": true,
        "name": "Anzuelos",
        "slug": "anzuelos"
    }
},
```

Ej. Crear categoría padre e hija

```
{  
    "pk": 1,  
    "model": "catalogue.category",  
    "fields": {  
        "description": "",  
        "numchild": 6,  
        "slug": "anzuelos",  
        "depth": 1,  
        "path": "0001",  
        "image": "",  
        "name": "Anzuelos"  
    },  
    {  
        "pk": 2,  
        "model": "catalogue.category",  
        "fields": {  
            "description": "",  
            "numchild": 0,  
            "slug": "ino",  
            "depth": 2,  
            "path": "0001002",  
            "image": "",  
            "name": "Acero inox"  
        },  
    },  
},  
}
```

*** Una vez creada toda la integración de la tienda junto al CMS y nuestra aplicación Web (Gadifishing), en el apartado “Fase de pruebas y mejoras”, pasaremos a integrar el catálogo, customizar la tienda y realizar las mejoras/modificaciones pertinentes de acuerdo a nuestras necesidades.*

Instalación de Wagtail CMS

Antes de pasar a instalar Wagtail y a la creación del blog, voy a contar el funcionamiento que deseamos del blog.

El blog será administrado por una persona que se encargará de crear los posts. Ningún usuario tendrá permisos para postear a menos que el encargado del blog le haya otorgado el permiso. No queremos un blog para uso de los usuarios, queremos que el blog sea para dar publicidad a la tienda y aumentar sus ventas.

Los usuarios podrán comentar los posts con total libertad.

Para instalar Wagtail escribimos el comando:

CLI> pip install wagtail

Una vez instalado, añadimos los módulos de Wagtail a las aplicaciones del fichero “`settings.py`” (variable `INSTALLED_APPS`) de nuestro proyecto.

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.forms',
    'wagtail.contrib.redirects',
    'wagtail.embeds',
    'wagtail.sites',
    'wagtail.users',
    'wagtail.snippets',
    'wagtail.documents',
    'wagtail.images',
    'wagtail.search',
    'wagtail.admin',
    'wagtail.core',
    "wagtail.contrib.routable_page",
    ...
]
```

*** Para crear el blog posteriormente, instalaremos los siguientes paquetes (también será necesario añadirlos en la variable `INSTALLED_APPS`).*

```
CLI> pip install django-el-pagination
CLI> pip install modelcluster
CLI> pip install taggit
```

```
INSTALLED_APPS = [
    ...
    'el_pagination',
    'modelcluster',
    'taggit',
    ...
]
```

Añadimos los middlewares de Wagtail también el fichero “`settings.py`”.

```
MIDDLEWARE = [
    ...
    'wagtail.core.middleware.SiteMiddleware',
    'wagtail.contrib.redirects.middleware.RedirectMiddleware',
]
```

En el fichero “`urls.py`” del proyecto, añadimos las urls para el uso de Wagtail CMS:

```
from wagtail.admin import urls as wagtailadmin_urls
from wagtail.core import urls as wagtail_urls
from wagtail.documents import urls as wagtaildocs_urls

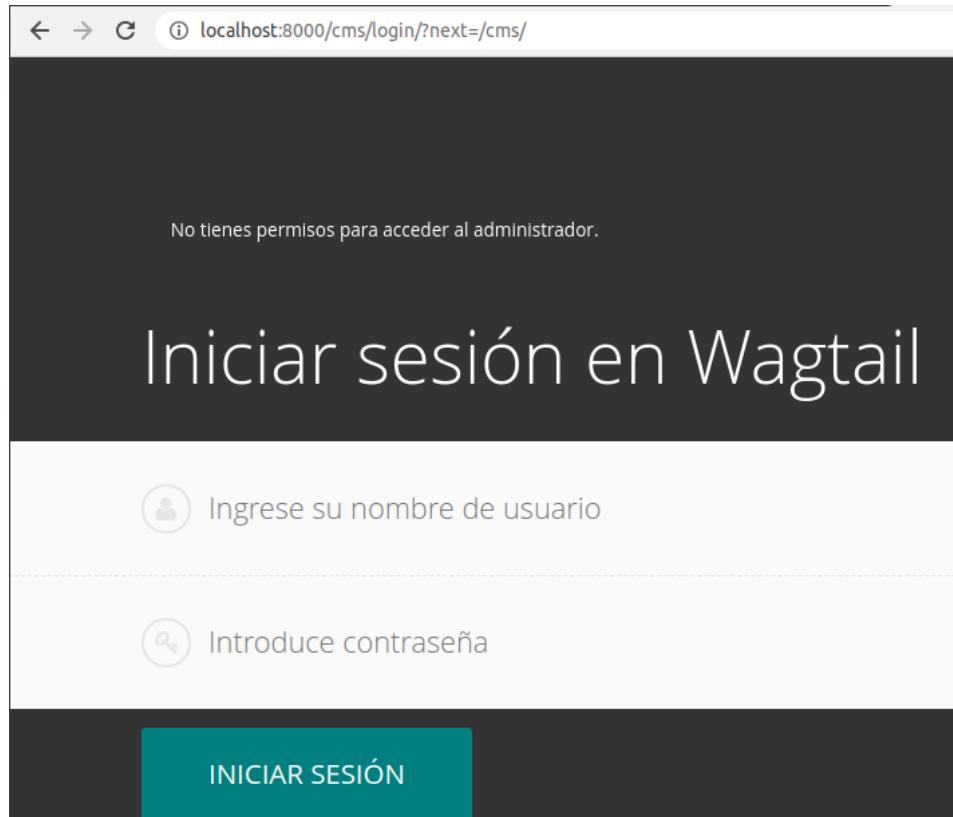
admin.autodiscover()

urlpatterns = [
    url(r'^cms/', include(wagtailadmin_urls)),
    url(r'^documents/', include(wagtaildocs_urls)),
```

Lo siguiente es realizar las migraciones para inicializar las tablas de Wagtail en nuestra base de datos:

CLI> python manage.py migrate

Ahora arrancamos nuestro server (`python manage.py runserver`), y podremos acceder al panel de administración de Wagtail CMS desde la url /cms.



Tras acceder con nuestras credenciales encontramos el panel principal:



Ya tenemos nuestro CMS preparado para crear nuestro Blog.

Creación del Blog

Para crear el blog, lo haremos añadiendo una nueva aplicación en nuestro proyecto. Luego integraremos esta aplicación en Wagtail para disponer de todas la ventajas de usar un CMS.

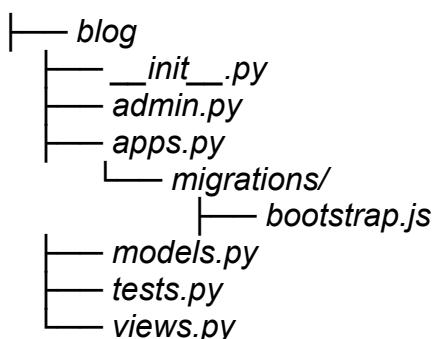
Creamos la aplicación llamada “blog” con el siguiente comando

```
CLI> python manage.py startapp blog
```

Luego, la añadimos en la variable INSTALLED_APPS.

```
INSTALLED_APPS = [  
    ...  
    'blog',  
    ...  
]
```

Tras crear la aplicación, se ha creado una estructura de directorios dentro del directorio blog como la siguiente:



Modificamos el fichero “`apps.py`” para definir la aplicación usada por Wagtail

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.apps import AppConfig

class BlogConfig(AppConfig):
    name = 'blog'
```

Ahora vamos a crear el cuerpo de la pagina principal del blog, el cuerpo de los posts del blog y los campos de nuestros formularios. Para ello añadimos en el fichero “`models.py`¹⁰” el siguiente código.

Lo siguiente es crear una vista para nuestra pagina principal del blog en el fichero “`views.py`”.

```
1 # -*- coding: utf-8 -*-
2 from __future__ import unicode_literals
3
4 from django.shortcuts import render
5
6 # Create your views here.
7 def inicio_blog(request):
8     return render(request, 'blog/blog_page.html')
```

Y creamos la url a esta vista en el fichero “`urls.py`”.

```
urlpatterns = [
    url(r'^cms/', include(wagtailadmin_urls)),
    url(r'^documents/', include(wagtaildocs_urls)),
    url(r'^blog/', include(wagtail_urls), name='inicio_blog'),
```

Ahora creamos un directorio para las plantillas del blog y creamos una plantilla “`base.html`” con el siguiente contenido:

```
{{ page.title }}
{% for entry in blog_entries %}
    {{ entry.title }}
{% endfor %}
```

10 Model.py blog:
https://github.com/imspaiK/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/blog/models.py

Creamos la plantilla de la pagina principal del blog “*blog_page.html*” con el siguiente contenido:

```
{% extends "blog/base.html" %}  
{% load static wagtailcore_tags wagtailimages_tags blogapp_tags el_pagination_tags  
wagtailmd %}  
{% block content %}  
{% endblock %}
```

Ahora si accedemos a la url /blog, podemos observar la pagina principal del blog

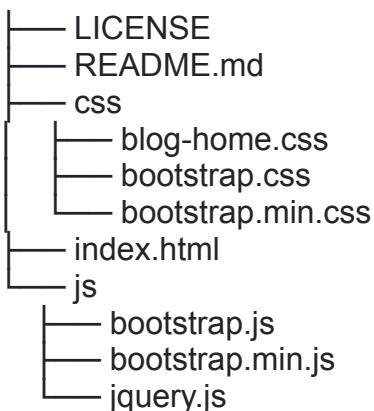


El siguiente paso será customizar nuestras plantillas y comenzar a crear los posts.

Customización del Blog

Para la customización del blog, usaremos un tema predefinido¹¹ de bootstrap

Descarguemos el archivo de tema y echemos un vistazo a lo que hay en el archivo, aquí está la estructura del proyecto bootstrap:



Ahora comenzamos a importar estos archivos en nuestro proyecto de blog wagtail, para convertir nuestra aplicación de blog en una aplicación Django reutilizable, prefiero colocar estos archivos en el directorio del proyecto.

Creamos una carpeta llamada blog dentro del directorio de los ficheros estáticos de nuestro proyecto y copiamos las carpetas “css” y “js”.

¹¹ Tema para blog: <https://startbootstrap.com/templates/blog-home/>

Añadimos al template base el código css y los ficheros javascript.

```
{% load static wagtailcore_tags wagtailimages_tags blogapp_tags %}

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

<script src="{% static 'blog/js/jquery.js' %}"></script>

<script src="{% static 'blog/js/bootstrap.min.js' %}"></script>
```

Creamos una plantilla para el contenido de cada post llamada “*post_page.html*” y añadimos:

```
{% extends "blog/base.html" %}
{% load static wagtailcore_tags wagtailimages_tags blogapp_tags wagtailmd %}

{% block title %}{{ self.title }} | {{ blog_page.title }}{% endblock title %}
{% block meta_title %}{% if self.seo_title %}{{ self.seo_title }}{% else %}{{ self.title }}{% endif %}
{% block meta_description %}{% if self.search_description %}{{ self.search_description }}{% else %}{{ block canonical %}}{% canonical_url self %}{% endblock canonical %}

{% block content %}
<br>
<br>
    {% if post.header_image %}
        {% image post.header_image original as header_image %}
        </img>
        <hr>
    {% endif %}
    <h1>{{ post.title }}</h1>

    <p>
        <i class="fa fa-clock"></i> {{post.date}} &ampnbsp
        <i class="fa fa-user"></i> {{post.owner}} &ampnbsp
        {% post_categories %}
    </p>
    <hr>

    {{ post.body|markdown|safe }}
    <hr>

    {% post_tags_list %}

    {% show_comments %}

{% endblock %}
```

Ahora añadimos el siguiente código a nuestra plantilla “*blog_page.html*” para mostrar correctamente cada post.

```
{% extends "blog/base.html" %}
{% load static wagtailcore_tags wagtailimages_tags blogapp_tags el_pagination_tags wagtailmd %}

{% block content %}

    {% for post in posts %}
        <div class="card mb-4">
            {% if post.header_image %}
                {% image post.header_image original as header_image %}
                <a href="{% post_date_url post blog_page %}">
                    </img>
                </a>
            {% endif %}

            <div class="card-body">
                <h2 class="card-title">
                    <a href="{% post_date_url post blog_page %}">{{ post.title }}</a>
                </h2>
                <p class="card-text">
                    {% if post.excerpt %}
                        {{ post.excerpt|markdown|safe }}
                    {% else %}
                        {{ post.body|markdown|safe|truncatewords_html:70 }}
                    {% endif %}
                </p>
                <a href="{% post_date_url post blog_page %}" class="btn btn-primary">Read More &rarr;</a>
            </div>
            <div class="card-footer text-muted">
                Posted on {{ post.date }}
            </div>
        </div>
    {% endfor %}


```

Si es necesario, podemos crear componentes adicionales para que la estructura html sea fácil de administrar, por ejemplo, "sidebar.html", "header.html", etc. Si usamos esta dinámica, debemos incluir estos elementos con la etiqueta include.

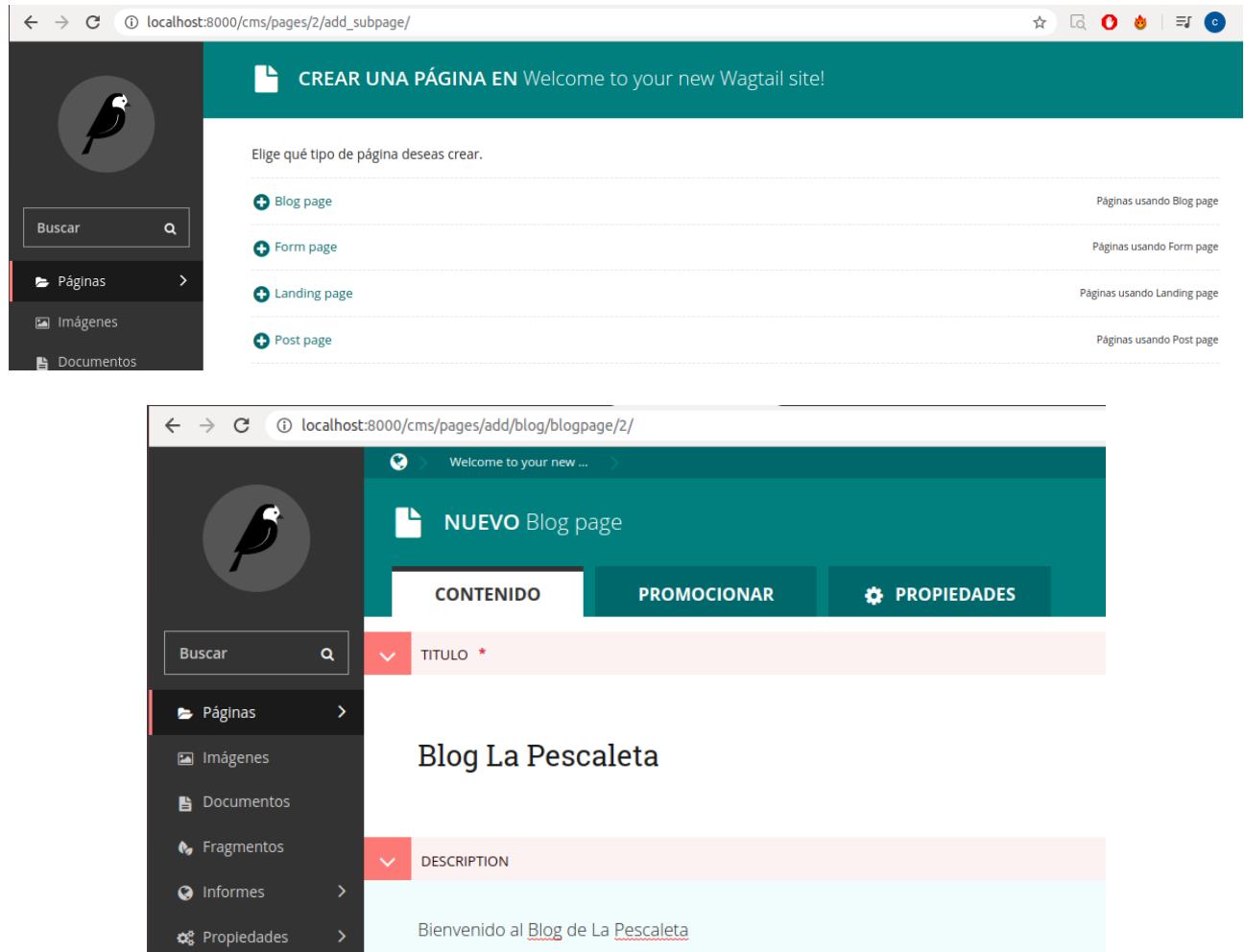
Por ejemplo: `{% include 'blog/components/sidebar.html' %}`

Por alguna razón, necesitamos hacer un trabajo adicional para que las categorías y las etiquetas funcionen como esperamos, ya que no hay una forma directa de mostrarlas en las plantillas.

Para ello creamos el directorio y fichero "`blog/templatetags/blogapp_tags.py`"¹².

En mi caso añado varios elementos en la plantilla base, como un paginador, categorías, etiquetas, links...

Ahora que ya tenemos nuestro blog personalizado¹³, accedemos al panel de Wagtail y creamos una página de blog.



The image consists of two screenshots of the Wagtail CMS interface. The top screenshot shows the 'Create New Page' page with a sidebar menu for Páginas, Imágenes, and Documentos. It lists four page types: Blog page, Form page, Landing page, and Post page, each with a count of pages using it. The bottom screenshot shows the 'Nuevo Blog page' creation form. It has tabs for CONTENIDO, PROMOCIONAR, and PROPIEDADES. The CONTENTO tab is active, showing fields for TÍTULO * (filled with 'Blog La Pescaleta') and DESCRIPTION (filled with 'Bienvenido al Blog de La Pescaleta').

12 `blogapp_tags.py`:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/blog/templatetags/blogapp_tags.py

13 Resultado final del blog:

https://github.com/imspaiik/documentacion_proyecto/tree/master/proyecto_aplicaciones_web/blog/templates/blog

Y luego añadimos páginas hijas como páginas de posts.

The screenshot shows a CMS interface for adding a blog post. The top navigation bar includes 'CONTENIDO' (Content), 'PROMOCIONAR' (Promote), and 'PROPIEDADES' (Properties). On the left, a sidebar menu lists 'Páginas', 'Imágenes', 'Documentos', 'Fragmentos', 'Informes', and 'Propiedades'. The main content area has a title 'El primero post' and a 'HEADER IMAGE' section containing a grid of small images related to seafood. The 'BODY' section contains the text 'Este es nuestro primer post'. A toolbar above the body section includes icons for bold, italic, heading, and other rich text options.

Y ahora podemos ver el resultado en la url /blog

The screenshot shows the published blog post 'El primero post' on a website. The post features a header image composed of nine smaller images of various seafood. Below the image is the title 'El primero post' and the text 'Este es nuestro primer post'. A blue button labeled 'Leer más →' is present. At the bottom, it says 'Publicado el 15 de Junio de 2020 a las 12:27'. To the right of the post is a sidebar with an orange header featuring the 'Gadifishing' logo (an anchor inside a circle with the text 'Gadifishing CADIZ'). The sidebar includes sections for 'Cristian Galnares Full Stack Developer', a search bar, categories ('No hay categorías'), and tags ('No hay etiquetas').

O bien el contenido del post.



A collage of nine images showing various types of seafood, including lobsters, crabs, mussels, oysters, and different shellfish.

Gadifishing
CADIZ
Infruta la pesca gaditana 2020

Cristian Galnares
Full Stack Developer

Buscar

Busqueda...
Buscar

Categorías

No hay categorías

Tags

No hay etiquetas

El primero post
15 de Junio de 2020 a las 12:27 spike

Este es nuestro primer post

0 Comentarios wagtail-tutorial Política de privacidad de Disqus 1 Acceder

Recomendar Tweet Compartir Ordenar por los mejores

Sé el primero en comentar...

Integración de Aplicación Web (Gadifishing)

Gadifishing es una aplicación Web basada en Python y Django, creada a lo largo del grado en la asignatura de IAW.

Esta aplicación sirve para llevar un registro de las capturas realizadas en nuestras salidas de pesca.

Es un proyecto independiente y lo que vamos a hacer es integrarlo como una aplicación en este proyecto.

Para integrarla junto a la tienda y el blog, debemos crear una nueva aplicación en el proyecto.

CLI> python manage.py startapp AppGF

Luego, la añadimos en la variable INSTALLED_APPS en el fichero “*settings.py*” del proyecto.

```
INSTALLED_APPS = [
...
    'AppGF',
    # Extras para Gadifishing
    'crispy_forms',
    'bootstrap4',
    'django_static_fontawesome',
]

```

Ahora clonamos el repositorio¹⁴ de Gadifishing

```
CLI> git clone https://github.com/imspaike/IAW
```

Dentro del archivo descargado, vamos a copiar el contenido de la carpeta AppGF y lo vamos a pegar dentro de la carpeta AppGF creada en nuestro proyecto.

Luego copiamos el directorio “/static/AppGF” y lo pegamos dentro del directorio de nuestro proyecto para los ficheros estáticos.

Y realizamos la misma operación con el directorio “/templates” y lo copiamos en nuestro directorio templates del proyecto.

Lo siguiente será habilitar las urls de Gadifishing, para ello añadimos en el fichero “*urls.py*” de nuestro proyecto:

```
urlpatterns = [
...
    path('AppGF/', include('AppGF.urls')),
    path('', include('AppGF.urls')),
]
```

Por ultimo, realizamos las migraciones y ejecutamos nuestro servidor para comprobar que todo funcione correctamente.

14 Repositorio de Gadifishing: <https://github.com/imspaike/IAW>

3. Fase de pruebas y mejoras

Con el despliegue del apartado anterior hemos conseguido la integración de las tres aplicaciones en un único proyecto común. Ahora utilizaremos esta unión como un proyecto único para realizar pruebas de funcionamiento, mejoras, customización...

Estilo común

Tras realizar unas pruebas iniciales de funcionamiento y comprobar que las tres aplicaciones no presenten errores, lo primero que realizo es la inserción de un estilo visual unificado, es decir, que todo el proyecto utilice el mismo estilo.

Para ello creamos una barra de navegación (navbar) y un footer para cada aplicación. Los incluimos en las plantillas base de las aplicaciones mediante la etiqueta `{% include %}`.

Ahora asignamos los colores para los elementos comunes a las tres aplicaciones (body, color de barra de navegación, color del footer, color de los textos...). Para ello modificamos los ficheros css que encontraremos en la carpeta “`static/nombreaplicación/css`” del proyecto

Barras de navegación (Navbar)

Vamos a utilizar dos tipos de barras de navegación: una única para la tienda y otra para los demás servicios (blog y Gadifishing) puesto que no queremos que aparezcan las categorías de nuestra tienda si estamos navegando por alguno de los servicios.

Navbar tienda

Primero vamos a crear una barra superior para los menús del usuario. Mostraremos distintas opciones dependiendo de si es usuario o administrador. Código completo¹⁵.

```
{% if user.is_authenticated %}
{% block nav_account_navbar_authenticated %}
<li>
    {% if num_unread_notifications > 0 %}
        # If user has new notifications - we change account link to go to inbox #
        <a href="{% url 'customer:notifications-inbox' %}">
            <i class="icon-user"></i>
            {% trans "Account" %}
            <span class="label label-warning">{{ num_unread_notifications }}</span>
        </a>
    {% else %}
        <a href="{% url 'customer:summary' %}"><i class="icon-user"></i> {% trans "Account" %}</a>
    {% endif %}
</li>
{% if user.is_staff or perms.partner.dashboard_access %}
    <li><a href="{% url 'dashboard:index' %}"><i class="icon-list-ul"></i> {% trans "Dashboard" %}</a></li>
{% endif %}
    <li><a href="{% url 'basket:summary' %}"><i class="fa fa-cart-arrow-down" style="font-size:30px" aria-hidden="true"></i>
        <li><a id="logout_link" href="{% url 'customer:logout' %}"><i class="icon-signout"></i> Salir</a></li>
    {% endblock %}
{% else %}
    <li><a id="login_link" href="{% url 'customer:login' %}"><i class="icon-signin"></i>Login/Registro</a></li>
    <li><a href="{% url 'basket:summary' %}"><i class="fa fa-cart-arrow-down" style="font-size:30px" aria-hidden="true"></i>
{% endif %}
```

Y luego crearemos una barra para navegador por las distintas secciones del portal.

15 Narvar tienda:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/templates/partials/nav_accounts.html

Haremos una barra dinámica en función de las categorías que tengamos en nuestra tienda, vamos a crear el siguiente código para evitar hardcodear el nombre de éstas categorías. Así en caso de modificar alguna categoría, no tendremos que modificar la plantilla.

```
<div class="collapse navbar-collapse" id="bs-megadropdown-tabs">
    <ul class="nav navbar-nav">
        <li class="active"><a href="{% url 'offer:list' %}" class="act">Inicio</a></li>
        <li class="active"><a href="{% url 'inicio' %}" class="act">Gadifishing</a></li>
        <li class="active"><a href="/blog" class="act">Blog</a></li>
    {% category_tree depth=2 as tree_categories %}
    {% if tree_categories %}
    {% for tree_category in tree_categories %}
    {% if tree_category.has_children %}
    <!-- Mega Menu -->
        <li class="dropdown">
            <a href="{{ tree_category.get_absolute_url }}" class="dropdown-toggle" data-toggle="dropdown">{{ t
            <ul class="dropdown-menu multi-column columns-3">
                <div class="row">
                    <div class="multi-gd-img">
                        <ul class="multi-column-dropdown">
                            </ul>
                    </div>
                </div>
                {% else %}
                <li class="dropdown">
                    <a href="{{ tree_category.get_absolute_url }}" class="active">{{ tree_category.name }}</a>
                {% endif %}
                {% for close in tree_category.num_to_close %}
                </li></ul>
            </li>
            {% endfor %}
            {% endfor %}
            {% endif %}
        </div>
    
```

Navbar servicios

En este caso, vamos a utilizar la misma barra para los menús de usuario/administrador y las secciones del servicio no van a cambiar. Código completo¹⁶.

```
<div class="collapse navbar-collapse" id="bs-megadropdown-tabs">
    <ul class="nav navbar-nav">
        <li class="active"><a href="{% url 'offer:list' %}" class="act">Inicio</a></li>
        <li class="active"><a href="{% url 'inicio' %}" class="act">Gadifishing</a></li>
        <li class="active"><a href="/blog" class="act">Blog</a></li>
        <li class="active"><a href="{% url 'pescados_list' %}" class="act">Pescados</a></li>
        <li class="active"><a href="{% url 'capturas_list' %}" class="act">Capturas</a></li>
    {% if request.user.is_authenticated %}
    <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">{{ user.username }} <i class="fa fa-angle-down"></i></a>
        <ul class="dropdown-menu">
            <li><a href="/usuario/detail/{{ user.pk }}>Perfil de pescador</a></li>
            <li><a href="{% url 'usuarios_update' user.pk %}>Editar perfil</a></li>
            <li><a href='/captura/detail/{{ user.username }}>Mis capturas</a></li>
            <li><a href="{% url 'capturas_create' %}>Añadir capturas</a></li>
            <li><a href="{% url 'logout' %}>Cerrar sesión</a></li>
        </ul>
    </li>
    {% else %}
    <li><a id="login_link" href="{% url 'customer:login' %}>Login/Registro</a></li>
    |{% endif %}
    </div>
```

16 Navbar servicios:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/templates/navbar.html

Footer

Con el footer realizamos la misma operación anterior, creamos un footer para la tienda y otro para el resto de servicios.

En el footer¹⁷ de la tienda incluiremos el siguiente código para mostrar las categorías pertenecientes a la tienda, ademas de ofrecer un banner con las marcas utilizadas.

```
<div class="col-md-3 w3_footer_grid">
    <h3>Categorías</h3>
    {% category_tree depth=1 as tree_categories %}
    {% if tree_categories %}
        {% for tree_category in tree_categories %}
            <ul class="info">
                <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="{{ tree_category.get_absolute_url }}">
                    {% for close in tree_category.num_to_close %}
                        ...
                    {% endfor %}
                    ...
                    ...
                {% endif %}
            </ul>
        {% endfor %}
        ...
    {% endif %}
</div>
<div class="col-md-3 w3_footer_grid">
    <h3>Perfil</h3>
    <ul class="info">
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="#">Perfil</a></li>
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="#">Carrito</a></li>
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="#">Login/Registro</a></li>
    </ul>
</div>
...
...
```

Por otro lado, el footer¹⁸ de los servicios, solo muestra la secciones que pertenecen a ellos.

```
<div class="col-md-3 w3_footer_grid">
    <h3>Secciones</h3>
    <ul class="info">
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="{% url 'pescados_list' %}">Pescados</a></li>
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="{% url 'capturas_list' %}">Capturas</a></li>
        <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="/blog">Blog</a></li>
    </ul>
</div>
<div class="col-md-3 w3_footer_grid">
    <h3>Perfil</h3>
    <ul class="info">
        {% if request.user.is_authenticated %}
            <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href='/usuario/detail/{{ user.pk }}'>Perfil de pescador</a></li>
            <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href='/captura/detail/{{ user.username }}'>Mis capturas</a></li>
            <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="{% url 'logout' %}">Cerrar sesión</a></li>
        {% else %}
            <li><i class="fa fa-arrow-right" aria-hidden="true"></i><a href="{% url 'customer:login' %}">Login/Registro</a></li>
        {% endif %}
    </ul>
</div>
```

Importar catalogo

En apartados anteriores vimos como añadir nuestro catálogo desde ficheros .csv y .json. Ya dispongo de mi estructura¹⁹ creada y realizo las importaciones de los ficheros.

17 Footer tienda:

https://github.com/imspaike/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/templates/partials/footer.html

18 Footer servicios:

https://github.com/imspaike/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/templates/footer.html

19 Catálogo: https://github.com/imspaike/documentacion_proyecto/tree/master/proyecto_aplicaciones_web/Catalogo

Tras la inserción del catálogo y los atributos, descubro que las opciones de producto no se ajustan a mi modelo de tienda. Dispongo de artículos que van a requerir un precio y un stock dependiendo de su número, tamaño... Como por ejemplo los anzuelos y cañas, que su precio cambia dependiendo del numero y la longitud.

Para solventar este problema, usaremos variantes de los artículos para poder asignar un precio y un stock a cada número del producto.

Esto se lleva a cabo creando un producto padre que contará con los atributos generales y específicos del tipo de producto. De éste producto padre colgarán productos hijos con precio y stock individual.

El resultado tras crear las variantes es el siguiente:



Ahora el usuario puede seleccionar el numero y encontrará el precio y stock de cada variante y ademas podrá añadirlo al carrito. Esto se hace demasiado tedioso para un usuario inexperto, así que voy a crear un formulario para seleccionar la variante del producto y poder añadirlo directamente desde la misma ventana.

Modificamos la plantilla detalle²⁰ de productos y añadimos:

```
{% block variants %}
<form id="add_to_basket_form" action="{% url 'basket:add' pk=product.pk %}" method="post" class="add-to-basket">
    {% csrf_token %}
    {% include "oscar/partials/form_fields.html" with form=basket_form %}
    <input type="hidden" name="quantity" value="{{ product.pk }}" id="id_quantity">
    <div class="form-group">
        <label for="id_child_id">Número de anzuelo</label>
        <div class="form-control">
            <select name="child_id" id="fg_id" onchange="validarnivel()">
                {% for child in product.children.all %}
                    {% purchase_info_for_product request child as child_session %}
                    {% if child_session.availability.is_available_to_buy %}
                        <option name="variantes" value="{{ child_pk }}" data-precio="{{ child_session.price.excl_tax|currency:child_sess }}>
                    {% endif %}
                {% endfor %}
            </select>
        </div>
    </div>
    <button type="submit" class="btn btn-lg btn-primary btn-add-to-basket" value="{% trans "Add to basket" %}" data-loading-"/>
</form>
{% endblock %}
```

20 Detalles de producto:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/catalogue/templates/detail.html

Ahora el usuario dispone de un formulario de selección y un botón para añadir directamente al carrito.



Ahora añadiremos una función (*onchange*²¹) para mostrar en la plantilla el precio del producto cuando vamos seleccionando la variante:

```
{% block extrascripts %}
    {{ block.super }}
<script>
    function validarnivel(){
        valor = document.getElementsByName('variantes');
        for (var i=0; i<=valor.length; i++){
            if (valor[i].selected){
                canivel = valor[i].dataset.precio;
                document.getElementById("precio_variante").innerHTML = canivel;
            }
        }
    }
</script>
{% endblock %}
```

Mostrar impuestos

Como vemos en las imágenes anteriores, el precio mostrado no detalla si es con impuestos o no.

Por defecto Django Oscar no trae configurado el porcentaje de impuestos (debido a que varía en cada país). Debemos ajustarlo en el código fuente.

21 Función al final de la plantilla:

https://github.com/imspaik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/catalogue/templates/detail.html

Para ajustar el porcentaje de impuestos, modificamos el fichero “strategy.py”²².

Con esto conseguimos añadir un 21% al precio base de los productos.

Para mostrar los precios con impuestos incluidos, añadimos en la plantilla “stock_record.html”²³ lo siguiente:

```
{% if session.price.exists %}
    {% if session.price.excl_tax == 0 %}
        <p class="price_color" id="precio_variente">{{ trans "Free" }}</p>
    {% elif session.price.is_tax_known %}
        <p class="price_color" id="precio_variente">Precio con IVA {{ session.price.incl_tax|currency:session.price.currency }}</p>
        <p class="price_color" id="precio_variente2">Precio sin IVA {{ session.price.excl_tax|currency:session.price.currency }}</p>
    {% endif %}
    ...
}
```

Y ahora podremos ver dos precios en la plantilla de detalles del productos



También debemos modificar la función onselect de las variantes para que el precio con impuestos cambie.

```
{% block extrascripts %}
    {{ block.super }}
<script>
    function validarnivel() {
        valor = document.getElementsByName('variantes');
        for (var i=0; i<=valor.length; i++){
            if (valor[i].selected){
                canivel = valor[i].dataset.precio;
                canivel2 = valor[i].dataset.iva;
                document.getElementById("precio_variente").innerHTML = "Precio con IVA " + canivel;
                document.getElementById("precio_variente2").innerHTML = "Precio sin IVA " + canivel2;
            }
        }
    }
</script>
```

Añadir botón de cantidad

Una de las típicas cosas a la hora de hacer compras online, es poder seleccionar la cantidad de un mismo artículo que queremos comprar. Django Oscar por defecto, solo permite esta opción una vez el producto ha sido añadido al carrito.

22 Fichero strategy.py:

https://github.com/imspaike/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/partner/strategy.py

23 Plantilla stock_record.html:

https://github.com/imspaike/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/catalogue/templates/partials/stock_record.html

Esto hace que sea algo pesada la tarea de elegir el número de artículos a comprar, ya que el usuario debe dirigirse al carrito para modificar la cantidad.

Está comprando	Cantidad
820-SS -- Nº 3 Disponible	1 Actualizar Eliminar

Para solventar este problema, voy a crear un botón de cantidad y lo voy a añadir al un formulario anterior para validar la cantidad.

```
<label for="id_quantity">Cantidad:</label>
<input type="number" name="quantity" id="id_quantity" value="1" min="0">

<form id="add_to_basket_form" action="{% url 'basket:add' pk=product.pk %}" method="post" class="add-to-basket">
  {% csrf_token %}
  {% include "oscar/partials/form_fields.html" with form=basket_form %}
  <label for="id_quantity">Cantidad:</label>
  <input type="number" name="quantity" id="id_quantity" value="1" min="0">
  <div class="form-group">
    <br>
    <label for="id_child_id">Número de anzuelo</label>

    <select name="child_id" class="form-control" id="id_child_id" onchange="validarnivel()">
      {% for child in product.children.all %}
        {% purchase_info for product request child as child_session %}
        {% if child_session.availability.is_available_to_buy %}
          <option name="variantes" value="{{ child.pk }}" data-precio="{{ child_session.price.excl_tax|currency:child_session.price.currency }}>{{ child.name }}
        {% endif %}
      {% endfor %}
    </select>

  </div>
  <button type="submit" class="btn btn-primary btn-block" value="{% trans "Add to basket" %}" data-loading-text="{% trans 'Adding...'">
</form>
```

Y ahora modificamos el estilo a nuestro gusto. Mi resultado final ha sido:

Inicio / Anzuelos / 820-SS

820-SS

Precio con IVA €12,10

Precio sin IVA €10,00

✓ Disponible

Escribir un comentario

Cantidad:

-

+

Número de anzuelo

820-SS -- Nº 3 -- €12,10

Añadir al carrito

Pasarelas de pago

Django Oscar tiene un abanico de extensiones para poder implementar los sistemas de pago. Tras investigar más a fondo, descubro que todas éstas extensiones están desmantenadas y olvidadas a excepción de la extensión django-oscar-paypal.

Otro problema de estas extensiones es que cada paquete está diseñado para un único tipo de pago, lo cual implica mantener varias aplicaciones y puede ser un poco tedioso de administrar.

Tras encontrar numerosos problemas con las extensiones propias de Django Oscar, decido investigar los paquetes²⁴ que ofrece Django para pagos externos.

Aquí también me encuentro que estos paquetes están desactualizados y no siguen un mantenimiento estable a excepción de dos paquetes basados en Stripe. A partir de aquí decido indagar más a fondo.

Stripe es una compañía tecnológica que su software permite a individuos y negocios recibir pagos por Internet. Proporciona la infraestructura técnica, de prevención de fraude y bancaria necesaria para operar sistemas de pago en línea. Poseen compatibilidad con diversos lenguajes de programación y disponen de una API muy potente y bien documentada.

Tras realizar varias pruebas con Stripe, decido utilizarlo para mi tienda porque ofrece todo lo que necesito. Permite a los usuarios pagar con cualquier tipo de tarjeta de crédito y además se encarga de verificar los datos bancarios. Posee su propio panel de administración sobre las transacciones.

Integración de PayPal

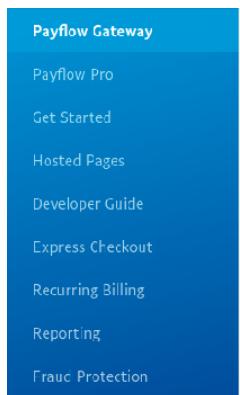
PayPal ofrece dos métodos de pagos: PayPal Express y PayPal PayFlow.

PayPal Express ofrece a los usuarios pagar a través de sus credenciales de PayPal. Nos permite pagar con saldo de la cuenta de PayPal o bien con una tarjeta de crédito asociada a ella.

PayPal Flow ofrece a los usuarios pagar a través de una tarjeta de crédito sin necesidad de ser cliente registrado en PayPal. Este método lo descartamos porque encontramos que no está disponible en España según su documentación²⁵.

24 Paquetes para realizar pagos con Django: <https://djangopackages.org/grids/g/payment-processing/>

25 Documentación PayPal Flow: <https://developer.paypal.com/docs/payflow/payflow-gateway/#>



Availability and Support

Create a free developer account [using these instructions](#). Or, sign up for a live Payflow Gateway account today. Select a country in which you conduct business from the following list:

- [United States](#)
- [Canada](#)
- [Australia](#)
- [New Zealand](#)

Payflow Gateway is available in the following countries. If your country does not appear in the list and you use Payflow without PayPal as your processor, Payflow is not available.

Para instalar esta extensión, usamos el comando:

```
CLI> pip install django-oscar-paypal
```

Una vez instalado, añadimos a las aplicaciones del fichero settings.py (variable INSTALLED_APPS) de nuestro proyecto lo siguiente.

```
INSTALLED_APPS = [  
    'paypal',  
    'paypal.express.dashboard.apps.ExpressDashboardApplication',  
    'paypal.payflow.dashboard.apps.PayFlowDashboardApplication',  
    'apps.shipping.apps.ShippingConfig',  
    'apps.checkout.apps.CheckoutConfig',  
    ...  
]
```

*** Con las dos ultimas líneas extendemos²⁶ las aplicaciones checkout y shipping propias de Django.*

Ahora debemos crear una API de pruebas en PayPal y conectarla con nuestra aplicación.

Para crear esta API, accedemos al panel²⁷ developer con las credenciales y creamos la API.

App name	Type	Actions
Default Application	REST	System generated, no actions available.
lapescaleta	REST	

26 Para saber más sobre Extender aplicaciones en Django Oscar, vea el ANEXO 2

27 Developer PayPal: <https://developer.paypal.com/developer>

Creamos una cuenta bussines y otra personal para poder testear nuestras compras.

Sandbox Accounts:

[Create bulk accounts](#)[Create account](#)

Total Accounts: 4

Account name	Type	Country	Date created	Manage accounts
javipaypal@personal.example.com	Personal	ES	09 Jun 2020	...
compralapescaleta@example.com	Business-Pro	ES	09 Jun 2020	...
sb-8003h2225359@personal.example.com	Personal	ES	09 Jun 2020	...
sb-s47vsp2228513@business.example.com	Business	ES	09 Jun 2020	...

Añadimos nuestras claves de la API al fichero “`settings.py`” de nuestro proyecto.

```
PAYPAL_API_USERNAME = 'compralapescaleta_api1.example.com'
PAYPAL_API_PASSWORD = 'tupasswd'
PAYPAL_API_SIGNATURE = 'tusignature'
```

Lo siguiente es añadir el botón de pago en nuestras plantillas.

```
{% block payment_details %}
    <div class="well">
        <div class="sub-header">
            <h3>Pagar con PayPal</h3>
        </div>
        <p>Click on the below icon to use Express Checkout but where the shipping address and method is already chosen on the merchant site.</p>
        <div style="overflow:auto"><a href="{% url 'paypal-direct-payment' %}" title="{% trans "Pay with PayPal" %}"></a></div>
    </div>
```

El resultado es:

Escriba los detalles del pago

Pagar con PayPal

Click on the below icon to use Express Checkout but where the shipping address and method is already chosen on the merchant site.



Y si pulsamos el botón de PayPal, nos redirecciona para realizar el pago mediante las credenciales.

Integración de Stripe

Ya contamos con que los usuarios pueden hacer pagos mediante PayPal o bien desde una tarjeta de crédito vinculada a PayPal.

Pero... ¿Y si el usuario quiere pagar con tarjeta y no dispone de cuenta PayPal?

Stripe nos ofrece esta solución. Gracias a su sistema de programación facade pattern²⁸.

Para poder utilizar Stripe solo necesitamos instalar el paquete, crear un fichero `facade.py`, y crear las vistas y formularios necesarios para enviar la información a su API.

CLI> pip install stripe

Accedemos a <https://stripe.com/es> y creamos una cuenta. Luego accedemos a nuestro panel de developer en <https://dashboard.stripe.com/> y obtenemos las claves de la API.

The screenshot shows the Stripe API Keys page. On the left, there's a sidebar with navigation links like Inicio, Activa tu cuenta, Pagos, Balances, Clientes, Cuentas conectadas, Productos, Informes, Desarrolladores (selected), Claves de API (selected), Webhooks, Eventos, Logs, and Configuración. A toggle switch for 'Viendo datos de prueba' is also present. The main content area has tabs for 'DATOS DE PRUEBA' and 'Más información sobre la autenticación de API'. Under 'Claves de API', it says 'Estás viendo las claves de API de prueba. Alternar para ver las claves activas.' A 'Viendo datos de prueba' button is shown. The 'Claves estándar' section lists two keys: 'Clave pública' with token 'pk_test_51Gsu0EYH9GArKzi3XIUoXj3M2Tjdeq8ELrF1Y0Rkyk4I9KgGH0ekphGIF0LU2ig6i8or5kVJCQ6CfRfDRpDh9w00R1Xe5dhh' and 'Clave secreta' with token 'sk_test_51Gsi'. The 'Claves restringidas' section shows 'No hay claves restringidas'. There's a '+ Crear clave restringida' button.

Añadimos las credenciales al fichero “`settings.py`” de nuestro proyecto y la moneda a utilizar.

```
STRIPE_SECRET_KEY="sk_test_51Gsi  
STRIPE_PUBLISHABLE_KEY="pk_test_  
STRIPE_CURRENCY = "EUR"
```

Creamos un fichero llamado “`facade.py`²⁹” que se encargará de enviar la información bancaria del pago a la API³⁰ de Stripe para que pueda o no, verificar el pago.

28 Para saber más sobre facade pattern: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

29 Fichero `facade.py`:
https://github.com/imspaik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/checkout/facade.py

30 Documentación API Stripe: <https://stripe.com/docs/api/authentication>

Dentro del directorio de nuestra aplicación extendida³¹ checkout añadimos el siguiente código al fichero “views.py³²”.

Este código se encarga de enviar la información del pago al panel de Stripe con el resultado de la operación bancaria.

Creamos el fichero “forms.py” y añadimos:

```
from django import forms
```

```
class StripeTokenForm(forms.Form):
    stripeEmail = forms.EmailField(widget=forms.HiddenInput())
    stripeToken = forms.CharField(widget=forms.HiddenInput())
```

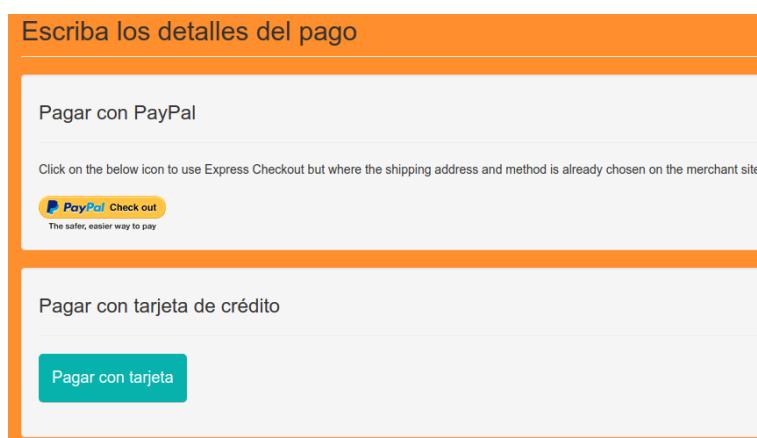
Y por último inicializamos la aplicación, añadiendo al fichero “__init__.py”:

```
PAYMENT_EVENT_PURCHASE = 'Purchase'
PAYMENT_METHOD_STRIPE = 'Stripe'
STRIPE_EMAIL = 'stripeEmail'
STRIPE_TOKEN = 'stripeToken'
```

Ya está todo listo para conectar y enviar los datos del pago a la API de Stripe, solo debemos añadir el formulario a nuestra plantilla³³ de método de pago.

```
<div class="well">
    <div class="sub-header">
        <h3>Pagar con tarjeta de crédito</h3>
    </div>
    <form action="{% url 'checkout:preview' %}" class="form-stacked" method="POST">
        <script src="https://checkout.stripe.com/checkout.js" class="stripe-button"
            data-key="{{ stripe_publishable_key }}"
            data-amount="{{ order_total_incl_tax_cents }}"
            data-name="La Pescaleta"
            data-description="{{ basket.num_items }} items {{ order_total.incl_tax|currency }}"
        </script>
    </div>
```

Y el resultado es:



31 Para saber más sobre Extender aplicaciones en Django Oscar, vea el ANEXO 2

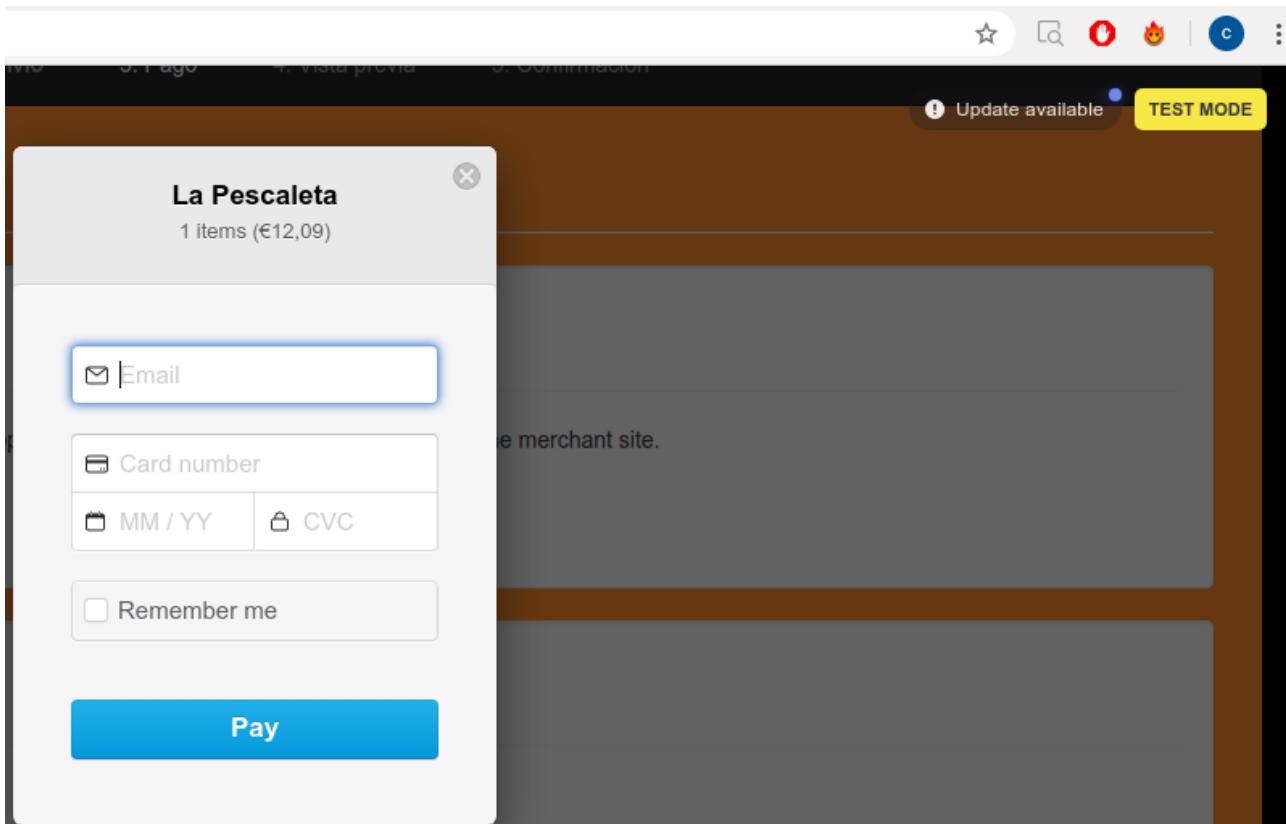
32 Fichero views.py:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/checkout/views.py

33 Resultado plantilla de pagos:

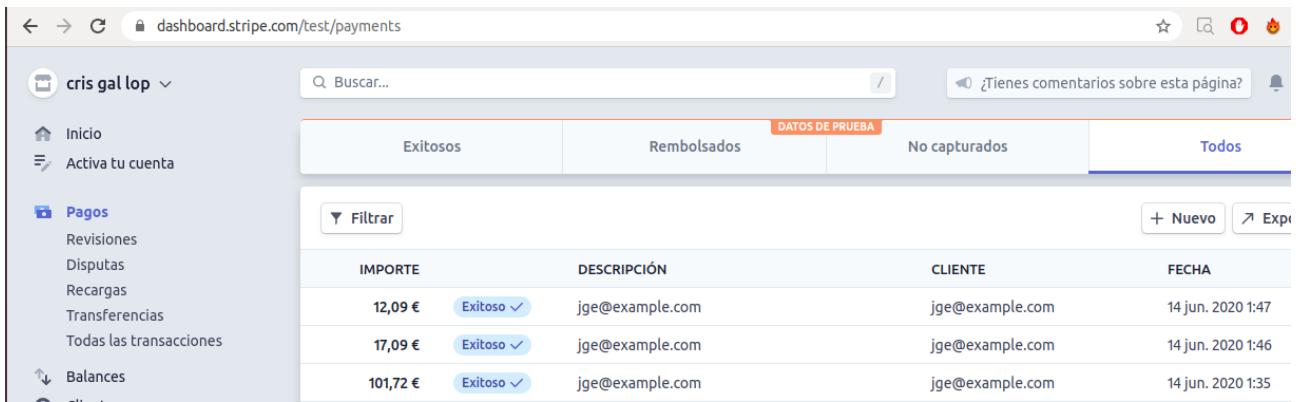
https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/apps/checkout/templates/payment_details.html

Tras pulsar el botón de pagar con tarjeta, nos aparece una ventana de Stripe para verificar los datos bancarios.



Y si los datos son aceptados, continuas para confirmar la compra.

Una vez realizado el pago y la compra, podemos acceder al panel de Stripe y comprobar el estado de las transacciones.



IMPORTE	DESCRIPCIÓN	CLIENTE	FECHA
12,09 €	Exitoso ✓	jge@example.com	14 jun. 2020 1:47
17,09 €	Exitoso ✓	jge@example.com	14 jun. 2020 1:46
101,72 €	Exitoso ✓	jge@example.com	14 jun. 2020 1:35

Conexión entre Gadifishing y tienda online

Como hemos ido comentando, queremos que los usuarios que utilicen Gadifishing para registrar sus capturas y tengan a su disposición los productos de la tienda para añadirlos en sus registros, en caso de haber utilizado artículos de ella.

Para lograr esto, debemos añadir a los modelos³⁴ de captura de Gadifishing, los campos deseados. En este caso caña, carrete y señuelo.

```
from oscar.apps.catalogue.models import Product
```

```
class Captura(models.Model):
    fecha_captura = models.DateField("Fecha DD/MM/YYYY")
    pescado = models.ForeignKey(Pescado, on_delete=models.CASCADE)
    cantpes = models.IntegerField(default=1)
    señuelo = models.ForeignKey('catalogue.Product', on_delete=models.CASCADE)
    carrete = models.ForeignKey('catalogue.Product', on_delete=models.CASCADE, related_name='carrete')
    caña = models.ForeignKey('catalogue.Product', on_delete=models.CASCADE, related_name='caña')
    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE)
    zona = models.ForeignKey(ZonaPesca, on_delete=models.CASCADE)
    marea = models.ForeignKey(Marea, on_delete=models.CASCADE)
    foto_captura = models.ImageField(upload_to='capturas', blank=True)

    class Meta:
        ordering = ['-fecha_captura', 'pescado']

    def __str__(self):
        return str(self.fecha_captura)
```

Añadimos los mismo campos a nuestro formulario³⁵ para listar al usuario los productos de cada tipo.

```
from oscar.apps.catalogue.models import Product
```

```
class CrearCapturaForm(forms.Form):
    fecha_captura = forms.DateField(help_text="Fecha DD/MM/YYYY")
    pescado = forms.ModelChoiceField(queryset=Pescado.objects.all())
    cantidad = forms.IntegerField()
    carrete = forms.ModelChoiceField(queryset=Product.objects.filter(product_class=3))
    caña = forms.ModelChoiceField(queryset=Product.objects.filter(product_class=2))
    señuelo = forms.ModelChoiceField(queryset=Product.objects.filter(product_class=4))
    zona = forms.ModelChoiceField(queryset=ZonaPesca.objects.all())
    marea = forms.ModelChoiceField(queryset=Marea.objects.all())
    foto_captura = forms.FileField(required=False)
```

34 Fichero models.py Gadifishing:

https://github.com/imspaik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/models.py

35 Fichero forms.py Gadifishing:

https://github.com/imspaik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/forms.py

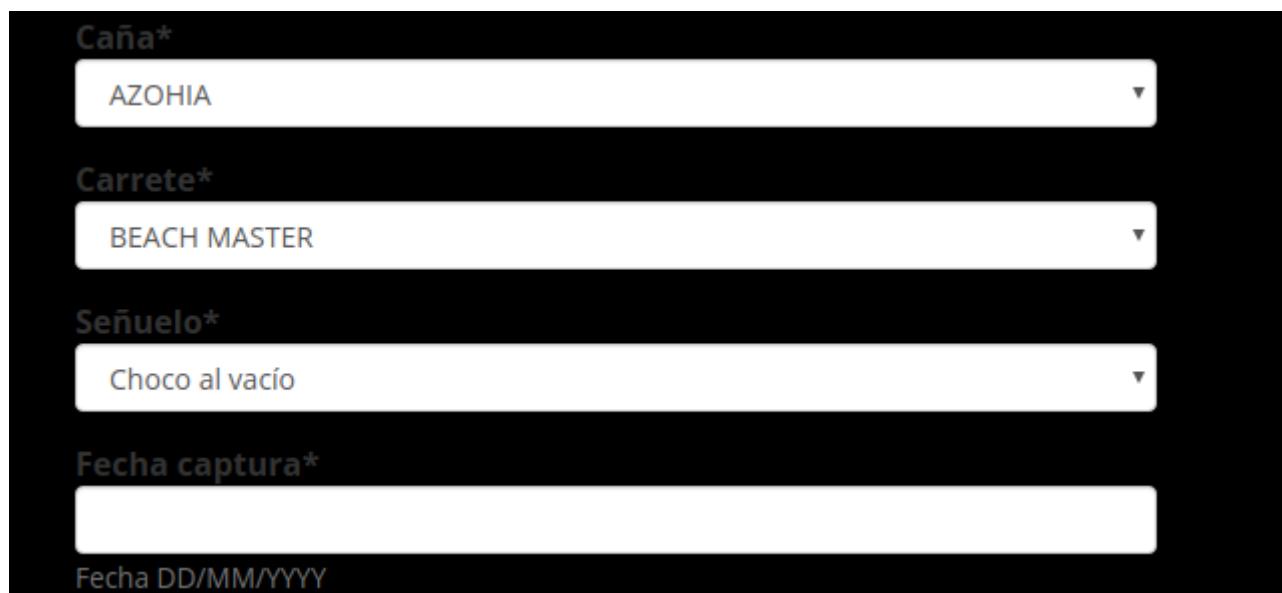
Y por ultimo también añadimos los campos a la función de validación en el fichero “views.py³⁶”:

```
def crearCaptura(request):
    if request.method == 'POST':
        form1 = CrearCapturaForm(request.POST, request.FILES)
        if form1.is_valid():
            capturas = Captura()
            usuario = Usuario.objects.get(user=request.user)
            capturas.usuario = usuario
            capturas.fecha_captura = form1.cleaned_data.get('fecha_captura')
            capturas.pescado = form1.cleaned_data.get('pescado')
            capturas.cantpes = form1.cleaned_data.get('cantidad')
            capturas.caña = form1.cleaned_data.get('caña')
            capturas.carrete = form1.cleaned_data.get('carrete')
            capturas.señuelo = form1.cleaned_data.get('señuelo')
            capturas.foto_captura = form1.cleaned_data.get('foto_captura')
            capturas.zona = form1.cleaned_data.get('zona')
            capturas.marea = form1.cleaned_data.get('marea')

            capturas.save()
            return HttpResponseRedirect('../..../captura')
    else:
        form1 = CrearCapturaForm()
    return render(request, 'AppGF/capturas.html', {'capturas': form1})
```

Modificamos la plantilla de capturas³⁷ para mostrar los nuevos campos añadidos.

Ahora cuando el usuario registra una captura, puede seleccionar los productos de nuestra tienda.



36 Fichero views.py Gadifishing:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/views.py

37 Plantilla capturas:

https://github.com/imspaiik/documentacion_proyecto/blob/master/proyecto_aplicaciones_web/AppGF/templates/AppGF/capturas_list.html

Y desde la vista general de la captura, encontramos enlaces directos a los productos

El pescado capturado es Mojarra

Detalles de la captura



Fecha: 5 de Mayo de 2020
Lugar de captura: Alameda
Pescador: superuser
Cantidad: 8
Caña utilizada: [AZOHIA](#)
Señuelo utilizado: [Navajas](#)
Carrete utilizado: [CM](#)

Cambiar la base de datos

Por defecto, la base de datos de todos los proyecto Django es una base de datos SQLite. Esta base de datos es recomendable para un entorno de pruebas, pero no para un entorno de producción.

Puesto que vamos a almacenar una gran cantidad de información proveniente de tres aplicaciones, necesitamos una base de datos mas potente, segura e independiente.

En mi caso he elegido Postgres³⁸ pero podemos cambiar con cualquier base de datos compatible³⁹ con Django.

Para poder utilizar Postgres como base de datos necesitamos de algunos requerimientos.

```
CLI> sudo apt-get install python-dev libpq-dev  
CLI> sudo apt-get install postgresql
```

Con esto se instalará Postgres y empezará a correr el servicio. Una vez que termine la instalación, le asignamos un password al usuario postgres.

```
CLI> passwd postgres
```

Añadimos un password y ahora ya podemos ingresar a ese usuario.

³⁸ Para saber más sobre Postgres: <https://www.postgresql.org/>

³⁹ Más informacion sobre bases de datos compatibles: <https://docs.djangoproject.com/es/3.0/faq/models/>

Creando una Base de Datos

Cuando ya estas adentro del usuario Postgres ya puedes empezar a crear lo que necesitamos que son tres cosas: una base de datos, un usuario de la base de datos, y asignar un password a ese usuario de base de datos.

```
CLI> createdb Ejemplo_DB  
CLI> createuser Ejemplo_Usuario  
CLI> psql  
CLI> ALTER USER Ejemplo_Usuario WITH PASSWORD 'password_usuario';
```

Con la primera linea creamos una base de datos con el nombre "Ejemplo_DB", con la siguiente linea creamos el usuario "Ejemplo_Usuario", con la siguiente linea ingresamos a la shell interactiva de Postgres para asignarle un password a nuestro usuario, en este caso es "password_usuario".

Psycopg2

Para conectarnos con cualquier motor de base de datos siempre necesitamos un conector, en este caso para conectar a Postgres con Python utilizamos Psycopg2, así que vamos a instalarlo dentro del entorno virtual.

```
CLI> source ./entorno/bin/activate  
CLI> pip install psycopg2
```

Configuración de Django

Para configurar Django vamos a editar el archivo "settings.py" de nuestro proyecto quedando de la siguiente manera:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql_psycopg2',  
        'NAME': 'proyecto',  
        'USER': 'proyecto',  
        'PASSWORD': 'tupasswd',  
        'HOST': '127.0.0.1',  
        'PORT': '',  
    }  
}
```

Ahora solo necesitamos realizar las migraciones a la base de datos.

Otras modificaciones

Ademas de estas modificaciones, también se han realizado otras de menor grado que pueden ser encontradas en el repositorio del proyecto.

Algunos ejemplos son: cambios de estilo, campos en formularios, modificaciones de plantillas...

4. Documentación del sistema

Ya conocemos un poco el funcionamiento de Django Oscar⁴⁰ y Wagtail CMS⁴¹, lo más recomendable para profundizar en ellos es acceder a sus propias documentaciones.

4.1 Manual de instalación.

Para hacer más simple el despliegue completo de todos los componentes del portal, se ha creado un repositorio con todo el proyecto. En este repositorio⁴² se han incluido todas las dependencias y configuraciones necesarias para su despliegue.

Una vez creada nuestra instancia⁴³ en AWS, clonamos el repositorio y seguimos las instrucciones que encontraremos en el fichero “README.md”.

Tras la instalación del repositorio, instalamos Apache, modulo wsgi y habilitamos el sitio en Apache.

```
CLI> sudo apt-get install apache2  
CLI> sudo apt-get install libapache2-mod-wsgi-py3
```

Para habilitar nuestro sitio, creamos un archivo de configuración para nuestra web en el directorio “/etc/apache2/sites-available”

```
CLI> sudo nano /etc/apache2/sites-available/"nombrenuestraweb.conf"
```

```
GNU nano 2.9.3 /etc/apache2/sites-enabled/lapescaleta.conf  
VirtualHost *:80>  
    ServerName lapescaleta.duckdns.org  
    #  
    Alias /static /home/webuser/www/django-oscar-master/sandbox/static/  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/sandbox/static>  
        Require all granted  
    </Directory>  
    #  
    Alias /static/ /home/webuser/www/django-oscar-master/src/oscar/static/  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/src/oscar/static>  
        Require all granted  
    </Directory>  
    Alias /media/ /home/webuser/www/proyectoasir/django-oscar-master/sandbox/public/media/  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/sandbox/public/media>  
        Require all granted  
    </Directory>  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/sandbox/public/static>  
        Require all granted  
    </Directory>  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/src/oscar>  
        Require all granted  
    </Directory>  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/src/oscar/static_src>  
        Require all granted  
    </Directory>  
    <Directory /home/webuser/www/proyectoasir/django-oscar-master/sandbox>  
        <Files wsgi.py>
```

40 Documentación Django Oscar: <https://django-oscar.readthedocs.io/en/2.0.4/index.html>

41 Documentación Wagtail CMS: <https://docs.wagtail.io/en/v2.9/index.html>

42 Repositorio del proyecto completo <https://github.com/imspaiik/proyectoasir>

43 En el Anexo 1 (Crear instancia AWS) se detalla la creación de la instancia.

Habilitamos nuestro sitio en apache

CLI> sudo a2ensite "nombrefichero.conf"

Recargamos el servicio del servidor apache2

CLI> sudo service apache2 reload

Lo siguiente es otorgar los permisos para que todo funcione correctamente.

Agregamos al usuario “webuser” al grupo “www-data” de apache2

CLI> sudo adduser webuser www-data

Modificar los permisos en la carpeta www

CLI> sudo chown -R webuser:www-data /home/webuser/www

Recolectamos los ficheros estáticos del proyecto

CLI> python manage.py collectstatic

Volvemos a establecer los permisos

CLI> sudo chown -R webuser:www-data /home/webuser/www

Volvemos a recargar apache2

CLI> sudo service apache2 reload

Por último comprobamos que apache arrancó correctamente

CLI> sudo service apache2 status

Añadir HTTPS

Agregar Certbot PPA

Deberá agregar el Certbot PPA a su lista de repositorios. Para hacerlo, ejecute los siguientes comandos en la línea de comandos de la máquina:

CLI> sudo apt-get install software-properties-common

CLI> sudo add-apt-repository universe

CLI> sudo add-apt-repository ppa:certbot/certbot

CLI> sudo apt-get update

Instalar Certbot

Ejecute este comando en la línea de comando en la máquina para instalar Certbot.

CLI> sudo apt-get install certbot python-certbot-apache

Elija cómo desea ejecutar Certbot

Opción 1: Para obtener un certificado y haga que Certbot edite su configuración de Apache automáticamente para servirlo, activando el acceso HTTPS en un solo paso.

CLI> sudo certbot --apache

Opción 2: Para obtener solo el certificado y añadirlo manualmente a apache

CLI> sudo certbot certonly --apache

Una vez generado el certificado, configuraremos las rutas en nuestro “fichero.conf”.

Renovación automática

Los paquetes Certbot en su sistema vienen con un trabajo cron o un temporizador systemd que renovará sus certificados automáticamente antes de que caduquen. No necesitará ejecutar Certbot nuevamente, a menos que cambie su configuración. Puede probar la renovación automática de sus certificados ejecutando este comando:

CLI> sudo certbot renew --dry-run

4.2 Manual de administración.

A continuación se expondrán procedimientos, distintos al proceso de “Instalación”. Tareas de administración como gestión del catálogo (añadir/productos, categorías...), gestión de pagos y cobros, gestión de pedidos, actualizaciones, etc..

Administración del servidor apache

Si realizamos algún cambio en las configuraciones del proyecto o bien en el propio servidor apache, debemos recargar el servidor web.

Para ello escribimos en la consola:

CLI> sudo service apache2 reload

En caso de problemas con el servicio apache disponemos de otros comandos para su administración que podemos consultar en su documentación⁴⁴.

Administración de la tienda online

Django Oscar dispone de un panel propio de administración para llevar a cabo todas las gestiones relacionadas con la tienda.

⁴⁴ Pagina Oficial de Apache2: <https://httpd.apache.org/docs/>

Para acceder a este panel, debemos dirigirnos a la url /dashboard y acceder con las credenciales de administrador.

A screenshot of a web browser displaying the Oscar CMS dashboard at localhost:8000/es/dashboard/. The page has a dark header with the 'oscar' logo, user information ('Bienvenido superuser@example.com'), and navigation links ('Volver a la tienda', 'Cuenta', 'Terminar sesión'). Below the header is a horizontal menu bar with items: Panel, Catálogo, Seguimiento, Clientes, Ofertas, Contenido, Informes, and PayPal. The 'Panel' item is highlighted with a blue background. The main content area has a light gray grid background and displays the text 'Panel'.

Como podemos ver, podemos movernos por las distintas secciones para llevar a cabo cualquier gestión relacionada con la tienda, como por ejemplo añadir productos, atributos, categorías...

Es un panel muy simple e intuitivo.

Ordenar índices del catalogo

En caso de realizar el borrado o inserción de objetos en nuestro catalogo, podemos encontrarnos con que los productos no llevan un orden numerado o bien hay saltos de orden. Por ejemplo tenemos 45 productos y al crear uno nuevo, se le asigna el numero 49.

Esto quiere decir que se han borrado algunos productos y al producto nuevo se le asigna el orden del ultimo producto creado.

Si queremos volver a ordenar la numeración, debemos indexar la base de datos. Para ello escribimos en la consola de la instancia:

```
CLI> python manage.py clear_index --noinput  
CLI> python manage.py update_index catalogue
```

Y volverá a haber un orden coherente y ordenado en nuestro catalogo

Poblar países

Podemos poblar la lista de países para nuestros formularios de manera automática

```
CLI> python manage.py oscar_populate_countries --initial-only
```

Importar catálogo

Como ya hemos visto en otras secciones anteriores, podemos crear nuestros catalogo desde el propio panel de la tienda o bien desde ficheros .csv y .json.

.CSV

```
CLI> python manage.py oscar_import_catalogue ruta/fichero.csv
```

.JSON

```
CLI> python manage.py loaddata ruta/fichero.json
```

Imágenes

CLI> python manage.py oscar_import_catalogue_images ruta/fichero.tar.gz

Administración Wagtail

Para administrar todo lo relacionado con nuestro blog disponemos del panel propio de Wagtail CMS.

Podemos encontrar el panel de administración en la url /cms.

Desde este panel podemos gestionar todos los contenido de nuestro blog y también podremos otorgar los permisos necesarios a los usuarios.

Es un panel simple e intuitivo.

4.3 Manual de usuario

Los usuarios disponen de una sección FAQ's donde encontraran ayuda sobre el portal.

5. Propuestas de mejoras

Desde mi punto de vista, este proyecto no está finalizado. En el estado actual, hay muchas puertas abiertas para continuar el desarrollo y mejora de éste.

Mejoras en Gadifishing

Pretendo añadir mas funcionalidades a este servicio, como tabla de mareas y vientos, extender el formulario de capturas y del perfil de pescador, añadir otras secciones como nudos, artes de pesca...

Añadir geolocalización para que los usuarios puedan añadir directamente su posicion de pesca.

Mejorar todo el estilo visual de las secciones, como por ejemplo, la forma en la que se ven las capturas y fotos.

Estudiar otras formas de conexión con la tienda, aparte de mediante las capturas.

Mejoras en la tienda

Mejorar el estilo visual de las categorías y productos.

Integrar el catalogo de productos directamente desde la web del proveedor

Integrar todos los pagos desde una unica plataforma (Stripe)

Añadir herramientas analíticas para realizar un estudio sobre qué mejoras podemos aplicar

6. Anexos

Anexo 1: Creación de Instancia en AWS

Accedemos a AWS con nuestra cuenta, nos movemos a servicios y seleccionamos EC2.

En la parte izquierda seleccionamos instancias.

The screenshot shows the AWS EC2 Dashboard. On the left sidebar, under the 'INSTANCES' section, 'Instances' is selected. The main area displays a summary of resources: 0 Running Instances, 0 Dedicated Hosts, 1 Volumes, 1 Key pairs, 0 Elastic IPs, 0 Snapshots, 0 Load balancers, 0 Placement groups, and 3 Security groups. A blue banner at the top says 'Welcome to the new EC2 console!'. Below it, a message states: 'We're redesigning the EC2 console to make it easier to use and improve performance. We'll release new screens try them and let us know where we can make improvements. To switch between the old console and the new c toggle.'

The screenshot shows the 'Instances' page under the 'INSTANCES' section. It lists a single instance: i-0a0943b2fd96d529e, which is an t2.micro type in us-east-1c, currently stopped. The 'Launch Instance' button is visible at the top of the page.

Una vez dentro, pulsamos sobre “Launch Instance” y nos mostrará la selección del sistema operativo.

The screenshot shows the 'Step 1: Choose an Amazon Machine Image (AMI)' step of the instance creation wizard. The search bar contains 'ubuntu'. The results list includes 'Ubuntu Server 18.04 LTS (HVM), SSD Volume Type - ami-04b9e92b5572fa0d1 (64-bit x86) / ami-0bb96c31d87e65d9 (64-bit Arm)', which is marked as 'Free tier eligible'. There are two radio buttons for architecture: '64-bit (x86)' (selected) and '64-bit (Arm)'. A 'Select' button is located to the right of the architecture options.

Elegimos en este caso Ubuntu 18.04 y pulsamos seleccionar.

Luego seleccionamos el tipo de instancia. En nuestro caso será t2.micro.

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your needs.

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)							
	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
<input type="checkbox"/>	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate
<input checked="" type="checkbox"/>	General purpose	t2.micro <small>Free tier eligible</small>	1	1	EBS only	-	Low to Moderate
<input type="checkbox"/>	General purpose	t2.small	1	2	EBS only	-	Low to Moderate
<input type="checkbox"/>	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate

El siguiente paso es añadir los detalles de la instancia. En nuestro caso los dejamos por defecto.

Step 3: Configure Instance Details

Step 3: Configure Instance Details
Configure the instance to suit your requirements. You can launch multiple instances from the same AMI, request Spot instances to take advantage of the lower pricing, assign an auto-role to the instance, and more.

Number of instances	<input type="text" value="1"/>	Launch into Auto Scaling Group
Purchasing option	<input checked="" type="checkbox"/> Request Spot instances	
Network	<input type="text" value="vpc-da520ba0 (default)"/>	Create new VPC
Subnet	<input type="text" value="No preference (default subnet in any Availability Zone)"/>	Create new subnet
Auto-assign Public IP	<input type="text" value="Use subnet setting (Enable)"/>	
Placement group	<input checked="" type="checkbox"/> Add instance to placement group	
Capacity Reservation	<input type="text" value="Open"/>	Create new Capacity Reservation
IAM role	<input type="text" value="None"/>	Create new IAM role
Shutdown behavior	<input type="text" value="Stop"/>	

Añadimos el disco duro.

Step 4: Add Storage

Step 4: Add Storage
Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. [Learn more](#) about storage options in Amazon EC2.

Luego pasamos a configurar las etiquetas, en este caso ninguna.

Y por último configuraremos el grupo de seguridad y permitimos el acceso mediante ssh.

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more about Amazon EC2 security groups.](#)

Assign a security group: Create a new security group
 Select an existing security group

Security group name:

Description:

Type <small>i</small>	Protocol <small>i</small>	Port Range <small>i</small>	Source <small>i</small>	Description <small>i</small>
SSH	TCP	22	Custom <small>v</small> 0.0.0.0/0	e.g. SSH for Admin Desktop

Finalizamos y tras lanzar la instancia, nos pedirá crear o usar un par de claves ssh.

Select an existing key pair or create a new key pair X

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

Choose an existing key pair v

Select a key pair v

I acknowledge that I have access to the selected private key file (paradjango.pem), and that without this file, I won't be able to log into my instance.

[Cancel](#) [Launch Instances](#)

Una vez creadas, se arranca la instancia y tras unos minutos podemos entrar mediante ssh.

Por último solo nos queda modificar el grupo de seguridad y añadir las excepciones HTTP y HTTPS.

Inbound rules					Edit inbound rules
Type	Protocol	Port range	Source	Description - optional	
HTTP	TCP	80	0.0.0.0/0	-	
HTTP	TCP	80	::/0	-	
Custom TCP	TCP	8000	0.0.0.0/0	-	
Custom TCP	TCP	8000	::/0	-	
SSH	TCP	22	0.0.0.0/0	-	
HTTPS	TCP	443	0.0.0.0/0	-	
HTTPS	TCP	443	::/0	-	

Extra: Añadir DNS a nuestra instancia

Para crear un DNS, vamos a la pagina <https://www.duckdns.org/> y creamos un subdominio para nuestra aplicación.

Una vez creado, añadimos una tarea programada a Cron para que cada 5 segundos compruebe la IP de la instancia y la vincule al subdominio que hemos creado.

1º Comprobamos si tenemos Cron instalado

```
CLI> ps -ef | grep cr[o]n
```

2º Instalamos curl

```
CLI> sudo apt-get install curl
```

3º Creamos un directorio y un script para nuestras configuraciones

```
CLI> mkdir duckdns  
CLI> cd duckdns  
CLI> nano duckdns.sh
```

Añadimos el siguiente contenido en el fichero sh echo

```
url="https://www.duckdns.org/update?domains=exampledomain&token=a7c4d0ad-114e-40ef-ba1d-d217904a50f2&ip=" | curl -k -o ~/duckdns/duck.log -K -
```

4º Cambiamos los permisos de este fichero

```
CLI> chmod 700 duck.sh
```

5º Configuramos cron

```
CLI> crontab -e
```

Y añadimos */5 * * * * ~/duckdns/duck.sh >/dev/null 2>&1

Ejecutamos nuestro script y listo

CLI> ./duck.sh

Extra: requisitos para aplicaciones Django

1.- Creamos la estructura para nuestro proyecto

Como usuario root creamos un nuevo usuario en llamado webuser

CLI> adduser webuser

Agregar el usuario a la lista de sudoers para poder ejecutar tareas como sudo

CLI> gpasswd -a webuser sudo

Cambiar al nuevo usuario creado webuser

CLI> su - webuser

Cambiar al home de webuser

CLI> cd ~

Crear un nuevo directorio para las webs

CLI> mkdir www

2.- Instalar virtualenv

Instalar el paquete pip en nuestro sistema.

CLI> sudo apt-get install python-pip

Instalar virtualenv en nuestro python

CLI> sudo pip install virtualenv

3.- Instalar Apache2 y modulo WSGI

Instalar apache

CLI> sudo apt-get install apache2

Instalar lib-wsgi para usar los proyectos de django

CLI> sudo apt-get install libapache2-mod-wsgi-py3

Anexo 2: Extendiendo aplicaciones de Django Oscar

Django Oscar está creado mediante aplicaciones independientes. Usa una aplicación para los cobros, los usuarios, el catálogo, el dashboard, los pagos...

Estas aplicaciones pueden ser extendidas o sustituidas por otras. En este caso, vamos a extender las aplicaciones de checkout y shipping para adaptarlas a los métodos de pagos elegidos.

Lo primero será añadir las aplicaciones al fichero “*settings.py*” (variable *INSTALLED_APPS*) de nuestro proyecto lo siguiente.

```
INSTALLED_APPS = [  
    'apps.shipping.apps.ShippingConfig',  
    'apps.checkout.apps.CheckoutConfig',  
]
```

Creamos dos apps en nuestro proyecto llamadas “*shipping*” y “*checkout*”. Una vez creadas, en la raíz del proyecto, creamos una carpeta llamada “*apps*” y movemos dentro sus directorios (*/apps/checkout* y */apps/shipping*).

En cada app, creamos o modificamos los siguientes ficheros quedando así:

/apps/checkout/apps.py

```
import oscar.apps.checkout.apps as apps  
class CheckoutConfig(apps.CheckoutConfig):  
    name = 'apps.checkout'
```

/apps/checkout/models.py

```
from oscar.apps.checkout.models import *
```

/apps/checkout/views.py

```
from oscar.apps.checkout.views import *
```

/apps/shipping/apps.py

```
import oscar.apps.shipping.apps as apps  
class ShippingConfig(apps.ShippingConfig):  
    name = 'apps.shipping'
```

/apps/shipping/models.py

```
from oscar.apps.shipping.models import *
```

/apps/shipping/views.py

```
from oscar.apps.shipping.views import *
```

7. Conclusiones finales

Tras estos casi 2 meses de proyecto utilizando Django he llegado a varias conclusiones:

Al principio, pensé que este proyecto iba a ser un poco más sencillo gracias a mis conocimientos de Django y de Python que he adquirido en estos años en el grado, pero la verdad es que no ha sido nada sencillo, ya que en el curso, todo el código utilizado ha sido creado por nosotros mismos desde cero. Ahora al trabajar con aplicaciones como Django Oscar y Wagtail CMS sin conocer absolutamente nada de ellas, ha sido un duro trabajo el aprender el funcionamiento de éstas.

Por otro lado esto me ha enseñado mucho sobre Django y Python, pero también he adquirido una gran variedad de conocimientos a la hora de realizar proyecto de este tipo. Por ejemplo la forma de estructurar el código, la jerarquía de directorios, importaciones y llamadas de funciones... También me ha enseñado (algo que no pensé decir jamás) que la integración de un CMS no es ninguna tontería, pese a que no me guste trabajar con ellos. Nos proporcionan una mayor gestión, administración... de todos los contenidos de nuestras webs.

Este proyecto también me ha enseñado como funcionan las pasarelas de pago por Internet. Además de haber descubierto Stripe, que es una gran plataforma en la que pienso investigar más y posiblemente utilizar en proyectos futuros de este tipo.

Para finalizar, pienso que se han cumplido todos los objetivos marcados para este proyecto y ha sido una gran experiencia académica que me ha abierto muchas posibilidades a otros proyectos futuros.

8. Bibliografía

Django

<https://docs.djangoproject.com/en/3.0/>

Django Oscar

<https://django-oscar.readthedocs.io/en/2.0.4/>

<http://oscarcommerce.com/>

<https://micropyramid.com/blog/how-to-customize-django-oscar-models-views-and-urls/>

Wagtail CMS

<https://docs.wagtail.io/en/stable/>

<https://www.accordbox.com/blog/wagtail-tutorials/>

PayPal

<https://developer.paypal.com/docs/express-checkout/#next-steps>

<https://developer.paypal.com/docs/payflow/express-checkout/#next-steps>

Stripe

<https://stripe.com/docs/payments#online>

<https://testdriven.io/blog/django-stripe-tutorial/>

<https://stripe.com/docs/api/authentication>