



ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

David Viejo Pomata

Grado de Ingeniería Informática

Consultor: Gregorio Robles Martínez

Profesor: Santi Caballe Llobet

Fecha: 10/1/2020



Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual [3.0 España de Creative Commons](https://creativecommons.org/licenses/by-nc-sa/3.0/es/)

A los que me han animado a estudiar.

FICHA DEL TRABAJO FINAL

Título del trabajo:	<i>ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER</i>
Nombre del autor:	<i>DAVID VIEJO POMATA</i>
Nombre del consultor/a:	Gregorio Robles Martinez
Nombre del PRA:	Santi Caballe Llobet
Fecha de entrega (01/2020):	10/01/2020
Titulación:	Grado de Informática
Área del Trabajo Final:	
Idioma del trabajo:	<i>Español</i>
Palabras clave	<i>Python flutter web kubernetes</i>
<p>Resumen del Trabajo (máximo 250 palabras): <i>Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.</i></p>	
<p>La mayoría de los equipos de desarrollo funcionan sin tener una arquitectura que les permita saber cómo tienen que hacer las tareas de desarrollo en cada momento. Este trabajo, está orientado a definir una arquitectura con el objetivo de facilitar la manera de trabajar de los programadores de un equipo</p> <p>El ámbito de esta arquitectura es el de desarrollo de aplicaciones web con Python y aplicaciones móviles en Flutter.</p> <p>Al definir esta arquitectura se prestará especial atención al role de programador, definiendo las condiciones de trabajo y herramientas que use para garantizar la mejor productividad, así como mejorar lo que se conoce como experiencia de desarrollador (DX Developer eXperience).</p> <p>Otro objetivo es de definir los procedimientos para que el desarrollo sea seguro ya que con el aumento de cibercrimen hay que pensar que el desarrollo tiene que cumplir reglas en cuanto a seguridad.</p> <p>Otro objetivo es el de definir los procedimientos para que la puesta en producción del software sea rápida y fiable ya que cada vez se pide a los equipos desplegar funcionalidad más rápidamente.</p> <p>Otro objetivo es el establecer procedimientos que garanticen la calidad y claridad del software. Hay que pensar que cada vez más es necesario incorporar nuevos programadores y que estos sean productivos cuanto antes y la calidad y claridad del software puede ayudar a esto.</p> <p>Otro objetivo es el de minimizar el número de errores en el software desarrollado. Dentro de la arquitectura debemos de definir los procedimientos para probar el</p>	

software. En producción debemos de usar herramientas que permitan detectar errores lo antes posible y así poder solucionarlos.

Otro objetivo es el de tener herramientas para definir varios entornos de producción de manera efectiva.

En cuanto a las aplicaciones móviles el objetivo es definir procedimientos de despliegue automatizados para los “store” de Android e IOS.

Como contexto de este trabajo definimos el realizar la arquitectura para un equipo de desarrollo de 10 personas con un responsable de desarrollo. Estos deben de desarrollar aplicaciones web con el framework Django y aplicaciones móviles con el framework flutter. La base de datos que han elegido es postgres. Los entornos de producción estarán en el Cloud de google soportados por el orquestador de contenedores kubernetes.

Abstract (in English, 250 words or less):

Most development teams operate without having an architecture that allows them to know how they have to do development tasks at all times. This goal of this work is aimed at defining an architecture with the goal of facilitating the way programmers in a team work.

The scope of this architecture is the development of web applications with Python using the framework Django and mobile applications in Flutter.

When defining this architecture, special attention will be given to the role of programmer, defining the working conditions and tools that he uses to guarantee the best productivity, as well as improving what is known as developer experience (DX Developer Experience).

Another objective is to define the procedures so that the development is safe since with the increase of cybercrime it is necessary to think that the development has to fulfill rules regarding security.

Another objective is to define the procedures so that the start-up of the software is fast and reliable since each time the teams are asked to deploy functionality more quickly.

Another objective is to establish procedures that guarantee the quality and clarity of the software. We must think that it is increasingly necessary to incorporate new programmers and that these are productive as soon as possible and the quality and clarity of the software can help this.

Another objective is to minimize the number of errors in the software developed. Within the architecture we must define the procedures to test the software. In production we must use tools that allow us to detect errors as soon as possible and thus be able to solve them.

Another goal is to have tools to define various production environments effectively.

As for mobile applications, the objective is to define automated deployment procedures for the Android and IOS store.

As a context of this work we define the realization of the architecture for a development team of 10 people with a development manager. They must develop web applications with the Django framework and mobile applications with the flutter framework. The

database they have chosen is Postgres. The production environments will be in the Google Cloud supported by the Kubernetes container orchestrator.

ÍNDICE

Contenido

1	Introducción	11
1.1	Contexto y justificación del Trabajo	11
1.2	Objetivos del trabajo	11
1.3	Enfoque y método seguido	12
1.4	Breve descripción de los otros capítulos de la memoria	12
2	Descripción de entornos	12
2.1	Entorno de desarrollo aplicaciones WEB	13
2.2	Entorno de desarrollo Flutter	14
2.3	Entorno de producción.....	15
2.4	Entorno de integración de código	15
2.5	Otros entornos.....	17
3	Normas de calidad del código Python.....	18
3.1	Análisis estático de código.....	18
3.2	Testing de código	18
3.3	Normas de codificación	18
3.4	Mejora en la codificación	18
3.5	Declaración de los tipos.....	19
3.6	Programación defensiva	20
3.7	Excepciones	20
3.8	Modularidad.....	22
4	Organización del código Django	23
4.1	La estructura de una aplicación Django	23
4.2	Creación de una aplicación web django.....	25
4.3	Los modelos	28
4.3.1	Los tipos de datos	29
4.3.2	Operaciones con el modelo	31
4.4	La lógica de negocio.....	33
4.5	Las vistas y los formularios.....	34
4.5.1	Relación entre urls y vistas	35
4.5.2	Tipos de Class-based views	35
4.5.3	Los formularios	37

4.5.4	Formularios basados en modelos	40
4.6	Las validaciones en los formularios.	41
4.7	Las templates	42
4.8	Apis con Django Rest Framework.....	44
4.9	Autenticación y la autorización.....	46
4.10	Generación de PDF	46
4.11	Almacenamiento de ficheros.....	46
4.12	El envío de Emails	46
4.13	La gestión de los errores	46
4.14	El caché.....	46
5	Proceso asíncrono de mensajes.....	47
5.1	Descripción del entorno	47
5.2	Celery con python.....	47
5.3	Celery con django.....	49
6	Descripción de aplicaciones Flutter.....	50
6.1	Widgets	50
6.2	Librerías para cada plataforma	55
6.3	Renderizado con SKIA	59
7	Integración y despliegue continuo (CI/CD).....	61
7.1	La integración continua.....	61
7.2	El despliegue continuo.....	63
7.3	Kubernetes como orquestador de contenedores	65
7.4	Despliegue de aplicaciones móviles	66
7.4.1	Desplegando en IOS	66
7.4.2	Desplegando en Android	69
8	Conclusiones	71
9	Glosario	72
10	Referencias.....	74

Lista de tablas

Tabla 1 Componentes de desarrollo	13
Tabla 2 Entorno de producción	15
Tabla 3 Integración de código.....	17
Tabla 4Tabla de componentes Django	24
Tabla 5Componentes Django	25
Tabla 6 Componentes de django	25
Tabla 7Tipos de campos.....	30

Lista de Ilustraciones

Ilustración 1 Entorno de desarrollo.....	13
Ilustración 2 Desarrollo Flutter	14
Ilustración 3 Estructura aplicación django	23
Ilustración 4Página inicio aplicación django	27
Ilustración 5 Error de ip no permitida	27
Ilustración 6 Relación modelo python y sql	28
Ilustración 7Esquema interacción modelo de negocio	33
Ilustración 8 Uso de formularios en django	40
Ilustración 9Sistema de templates de django	43
Ilustración 10 Funcionamiento de django rest framework	45
Ilustración 11Celery	47
Ilustración 12 Funcionamiento de celery	48

1 Introducción

1.1 Contexto y justificación del Trabajo

De acuerdo con el Project Management Institute (PMI) el 14% de los proyectos de IT fallan.

Las razones suelen ser principalmente por

- 1 Gestión no adecuada
- 2 Comunicación
- 3 Definición de alcance inadecuada
- 4 Gestión del cambio
- 5 Escalabilidad de la solución
- 6 Problemas en la gestión del software

La tecnología cada vez es más compleja. Los retos de hacer desarrollos seguros cada vez son más grandes. La necesidad de producir nueva funcionalidad en el menor tiempo posible cada vez es mayor.

El proceso de desarrollo, de generación de nuevas versiones del sistema, la puesta en producción de estas ha sido un proceso manual, en general. La necesidad de mejorar el sistema rápidamente hace que estos procesos tengan que ser automáticos y poder así responder a las necesidades del mercado

El trabajo pretende definir cómo deben de ser los entornos y los procedimientos para poder realizar desarrollos de calidad y seguros. El resultado a obtener es una descripción de una arquitectura y de los procedimientos para la automatización de tareas en el desarrollo y despliegue de aplicaciones.

Por contextualizar el problema a resolver, tenemos un equipo de desarrollo de entre 10 y personas que desarrollan aplicaciones web y móviles. Las aplicaciones web son desarrolladas con el framework **django** (1) y las aplicaciones móviles con el framework **flutter** (2) de Google.

Como requisito técnico, tenemos que las aplicaciones se han de desplegar en la nube en el entorno **Kubernetes** (3) .

El resultado que queremos obtener son las **guías** para que el equipo pueda realizar el desarrollo de software con calidad y seguridad, así como tener procedimientos de **integración continua** y de **despliegue continuo**.

1.2 Objetivos del trabajo

Como objetivo principal tenemos el definir una arquitectura basada usando los frameworks Django y Flutter. Estos framework permiten realizar aplicaciones web y aplicaciones móviles

El framework Django está basado en el lenguaje de programación Python, que durante este último año ha aumentado su uso, quizás debido a la conexión entre el Python y librerías de inteligencia artificial como Tensor Flow, Theano, Scikit-learn entre otras.

Por otro lado, Flutter cuya versión 1.0 data de diciembre de 2018, ha logrado muchos adeptos en la construcción de aplicaciones móviles multiplataforma. El lenguaje base de Flutter es el lenguaje Dart, creado por Google en 2011. Este lenguaje no triunfó

como sustituto de javascript, pero ha encontrado un hueco en la construcción de aplicaciones nativas para móviles.

Otros objetivos son:

1. Describir los entornos de desarrollo y producción usados.
2. Describir el framework para el desarrollo de aplicaciones móviles.
3. Describir el framework para el desarrollo de aplicaciones Django.
4. Definir buenas prácticas para realizar un desarrollo seguro y que no tenga vulnerabilidades.
5. Describir el procedimiento para desplegar aplicaciones en producción u otros entornos
6. Buenas prácticas para desarrollar código fácil de mantener y de testear.

1.3 Enfoque y método seguido

Para hacer este trabajo se ha contado con la experiencia de equipos de trabajo donde he estado trabajando.

Se han desarrollado pruebas de concepto de cada una de las partes de la arquitectura integrando diferentes productos y servicios. Todas las pruebas de concepto realizadas están en el repositorio de github: <http://github.com/dviejopomata/tfg>

Breve resumen de productos obtenidos

Repositorio con las pruebas de concepto realizadas en la construcción de la arquitectura situado en GitHub <http://github.com/dviejopomata/tfg>

1.4 Breve descripción de los otros capítulos de la memoria

En principio, describimos los entornos de desarrollo que existirán en el equipo. Tenemos en desarrollo dos entornos claramente diferenciados uno para hacer desarrollo web y otro entorno para desarrollo de aplicaciones móviles.

También se explican el entorno de producción y el los entornos de integración y despliegue continuo.

A continuación, describimos el lenguaje Python desde el punto de vista de la calidad del software

Pasamos a describir el Django, que el framework central para el desarrollo de aplicaciones web, por parte del equipo.

Prestamos atención al Celery, entorno Python de procesamiento de mensajes asíncrono. Estos entornos cada vez son más habituales en el desarrollo web.

Dedicamos un capítulo al desarrollo de aplicaciones móviles multiplataforma usando el entorno Flutter.

Por último, dedicamos un capítulo al despliegue de las aplicaciones tanto web como móviles en otros entornos. En este capítulo también abordamos las técnicas de integración de código de manera continua.

2 Descripción de entornos

Se trata de una arquitectura para desarrollar aplicaciones web Django y para el desarrollo de aplicaciones Flutter. Definiremos los entornos de desarrollo tanto para Django como para Flutter y el entorno de producción para las aplicaciones Django

2.1 Entorno de desarrollo aplicaciones WEB

En cuanto a los componentes que tiene que tener la máquina de desarrollo para el desarrollo con Django web server son:

MAQUINA DE DESARROLLO DJANGO

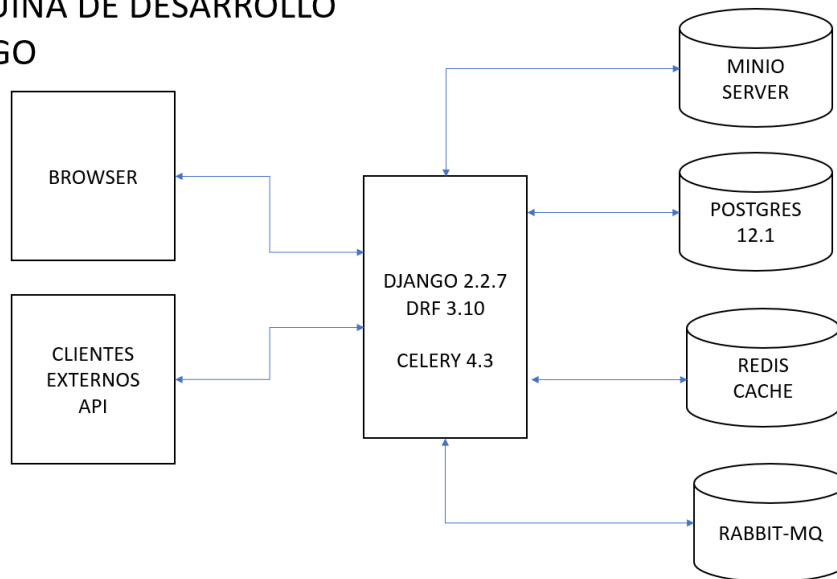


Ilustración 1 Entorno de desarrollo

Software instalado en la máquina de desarrollo

Browser	Navegadores para probar el sistema
Clientes externos	Postman para pruebas del api Curl para pruebas desde el Shell
Django, Celery y Django Rest Framework	Framework de trabajo para el desarrollo web, desarrollo de api y celery para trabajo de colas.
Minio	Servidor de ficheros
Postgres	Servidor de SQL
Rabbit-mq	Servidor de mensajes
MailHog	Simulador para comprobar el envío de mails.

Tabla 1 Componentes de desarrollo

En cuanto a las características hardware de la máquina de desarrollo

1. Dos pantallas para poder ver el código al mismo tiempo que el editor de código
2. La CPU evoluciona y se debe adquirir la más común que haya en el momento.
3. La memoria RAM es el punto más crítico del sistema. Debe de permitir ejecutar casi todas las aplicaciones en memoria. En la actualidad debería de tener al menos 32GB RAM.
4. En cuando al almacenamiento debería de ser de tipo SSD.
5. Si se necesitara más almacenamiento este debe de ser compartido con el resto del equipo.
6. Teniendo en cuenta criterios de seguridad, deberían de inhabilitarse el almacenamiento usb.

El software instalado en la máquina de desarrollo será

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

1. Editor de código que contemple el desarrollo en el framework Django. Usaremos Pycharm
2. Editor de código que contemple el desarrollo en el framework flutter.
3. Repositorio de código git para gestionar el código fuente.
4. Base de datos postgres como gestor de base de datos.
5. Sistema de almacenamiento de ficheros minio compatible con S3 de AWS.
6. Redis para almacenar las sesiones.
7. Celery para manejar el tratamiento de mensajes.

2.2 Entorno de desarrollo Flutter

En cuanto al entorno de apps con Flutter tenemos la siguiente figura

DESARROLLO CON FLUTTER

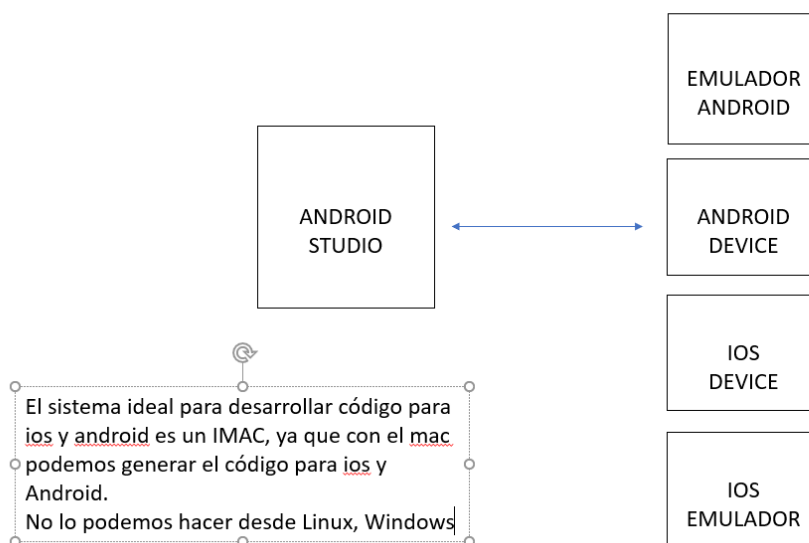


Ilustración 2 Desarrollo Flutter

Flutter es un entorno desarrollado por Google para el desarrollo de aplicaciones móviles multiplataforma. El lenguaje de desarrollo es el Dart, también desarrollado por Google.

Una vez desarrollada la aplicación esta se compila y se genera código para Android y código para IOS. Para ello es necesario disponer del Android Studio para poder hacer el apk de Android. También es necesario tener el Xcode de IOS para generar el código para pasarlo a dispositivos Apple.

Para la validación de la aplicación se usará preferentemente emuladores de los diferentes dispositivos. También será necesario disponer de dispositivos reales para hacer pruebas.

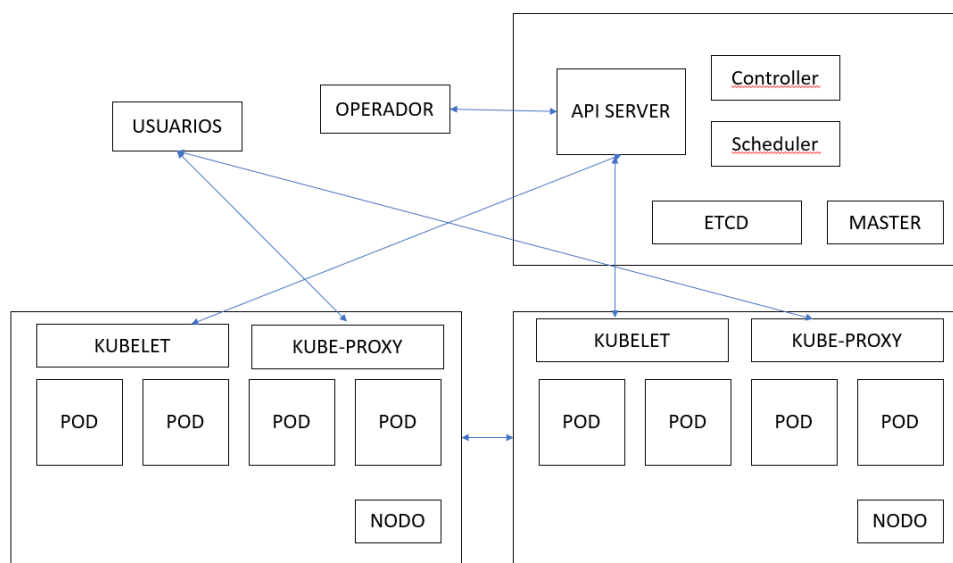
El proceso de actualizar las apps en los stores es un proceso que hay que automatizar. Hay que crear imágenes de la app y esto si se hace de manera manual puede llevar tiempo.

2.3 Entorno de producción

El entorno de producción está basado en la nube. El despliegue de las aplicaciones se hará usando contenedores Docker. Estos contenedores se orquestrarán usando kubernetes.

Kubernetes es infraestructura definida en código. Esto quiere decir que definiremos una serie de recursos y el sistema tratará de alcanzar esa definición. El elemento central del kubernetes es el POD, que es donde se ejecutan los contenedores.

Estos POD, pueden estar duplicados y repartidos por otro elemento fundamental que son los NODOS. Por lo tanto, cada nodo tendrá varios "pod". Existen otros elementos como los Services y los Ingres que definen como se llega a los elementos definidos en los "pods".



ENTORNO DE PRODUCCIÓN KUBERNETES

Tabla 2 Entorno de producción

2.4 Entorno de integración de código

El sistema de integración está centrado en la herramienta de código fuentes GITLAB, que contiene herramientas para definir las operaciones a realizar para hacer CI/CD Integración Continua, Despliegue Continuo.

Por integración continua se entiende el proceso de crear la nueva reléase a partir del código de cada uno de los componentes. Básicamente en este proceso se pasar las validaciones de código, se genera una build y a esta se pasan los test.

En esta fase los test son fundamentales, ya que un programador puede integrar un código que funciona en su máquina pero que es incompatible con el código actual del repositorio. El sistema de CI hará el proceso de validación y el paso de los test de manera automática.

Por otro lado, CD, despliegue continuo, trata de preparar el código para desplegarlo en el entorno de producción. No solo se usa para en entorno de producción. Un sistema puede tener varios entornos como prueba, formación, producción.

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

La tendencia en CD es lograr hacerlo de manera automática, previa ejecución sin errores de una serie de tareas orientadas a la validación de la versión.

Para hacer estos despliegues, utilizaremos los [runners de gitlab](#), los cuales nos permiten ejecutar código programáticamente basándonos en un gitlab-ci.yml. Este fichero nos permite ejecutar comandos bash/powershell en Windows, Mac y Linux.

Un ejemplo de gitlab-ci.yml en el que construiríamos la aplicación, testearíamos la aplicación y desplegaríamos es el siguiente:

```
1 stages:
2   - build
3   - test
4   - deploy
5
6
7 build_webapp:
8   script:
9     - [BUILD COMANDO]
10  stage: build
11
12 test_webapp:
13  script:
14    - [TEST COMANDO]
15  stage: test
16
17 deploy_app:
18  script:
19    - [DEPLOY COMANDO]
20  stage: deploy
```

He puesto placeholders en la sección de script, porque esto depende de las herramientas que utilizemos.

INTEGRACIÓN DE CÓDIGO

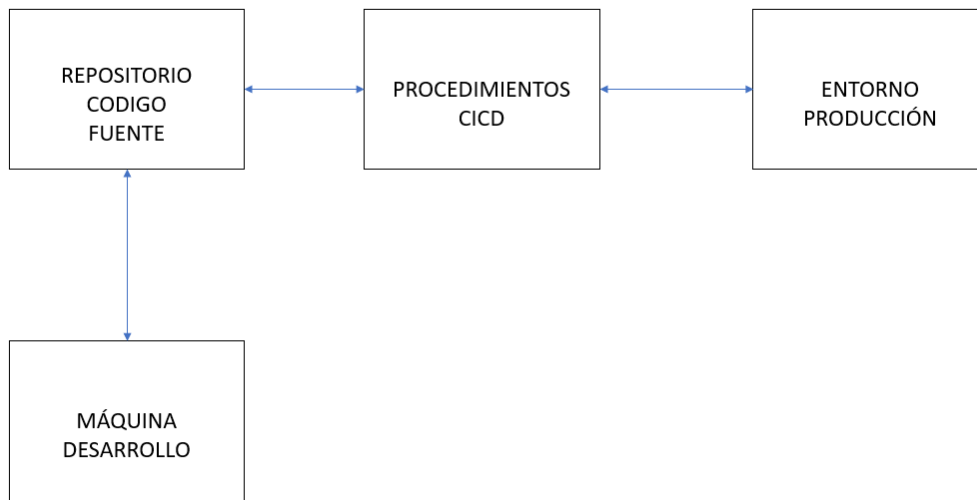


Tabla 3 Integración de código

2.5 Otros entornos

Es posible que dentro la explotación de una aplicación aparezca la necesidad de otros entornos.

Si hemos desarrollado un sistema, hay que enseñarlo y por lo tanto, tenemos que tener un entorno estable, lo más parecido al entorno de producción que permita conocer sus funcionalidades.

En un sistema en producción se producen errores. Estos errores hay que corregirlos y probarlos. Para probar que funciona correctamente el software corregido hay que usar un entorno con las mismas características que producción.

Es posible que nuestro sistema necesite ser probado con cargas parecidas a las de producción, con el fin de garantizar que el sistema tiene buenos tiempos de respuesta.

En general debemos de automatizar la creación de entornos de cualquier tipo, en el menor tiempo posible.

3 Normas de calidad del código Python

Python es un lenguaje de propósito general y existen en la actualidad alrededor de él muchos frameworks y librerías que ayudan en las tareas de codificación.

En la arquitectura de desarrollo que estamos definiendo el código es una parte principal y debemos de definir normas para que este sea claro y con la complejidad menor posible.

3.1 Análisis estático de código

Existen herramientas que se integran con los editores y que permiten que se realice un análisis estático del código. A éstas herramientas se les conoce con el nombre de "linters". Entre ellas se encuentran: pylint, pep8, bandit, etc.

Activaremos en el editor el linter pylint que nos permitirá marcar el código que no cumple con las reglas.

3.2 Testing de código

Los test son una parte importante en la calidad del software. Podemos afirmar que si un sistema no tiene test, no será fiable y sobre todo su mantenimiento será muy complejo.

Los test hacen que cuando se haga refactoring para mejorar el código, los test nos ayudarán a detectar los errores producidos por los cambios.

También cuando añadido nuevas funcionalidades que afectan a las antiguas los test nos ayudarán en la tarea de poner a punto el software.

Tenemos test unitarios y test de integración. Los test unitarios están orientados a probar el comportamiento de una determinada funcionalidad y los test de integración nos permiten probar la funcionalidad completa.

Existen muchos framework para hacer test. De entre ellos vamos a definir en esta arquitectura unittest que viene con la instalación de Python y está disponible sin instalar ningún paquete adicional.

3.3 Normas de codificación

En la codificación se seguirán las normas definidas en la guía de estilo PEP-8 de Python. Las reglas de esta guía están controladas por el editor, por lo tanto, al programador no le será complicado seguirlas.

3.4 Mejora en la codificación

Cuando estamos codificando muchas veces tenemos que trabajar con colecciones. Tenemos que iterar sobre colecciones obteniendo otra colección filtrando algunos elementos y transformando los elementos.

Por ejemplo, si no piden sumar el producto pvp por unidades en la lista de diccionarios siguiente, podemos codificarlo de la siguiente forma:

```
ventas = [  
    {"codigo":1, "pvp":6, "unidades":1},  
    {"codigo":2, "pvp":4, "unidades":2},
```

```
{ "codigo":3, "pvp":4, "unidades":5},  
{ "codigo":4, "pvp":2, "unidades":6}  
]  
total = 0  
for venta in ventas:  
    total+= venta["pvp"] * venta["unidades"]  
print(total)
```

pero se puede usar el bucle por una lista por compresión de la siguiente forma

```
total = sum([venta["pvp"] * venta["unidades"] for venta in ventas])
```

Si queremos obtener de la lista aquellas líneas de venta cuyas unidades sean mayores de 3, podemos hacer

```
mayores = [venta for venta in ventas if venta['unidades'] > 3]
```

en vez de

```
mayores = []  
for venta in ventas:  
    if venta['unidades'] > 3:  
        mayores.append(venta)
```

Como **recomendación** en la arquitectura, hay que evitar **for** e **if** para el tratamiento de **colecciones**.

3.5 Declaración de los tipos

En Python podemos declarar los tipos de parámetros y el tipo de resultado de las funciones.

Por ejemplo, la siguiente en el siguiente código

```
class Clase():  
    def __init__(self, p1: int, p2: int):  
        self.p1 = p1  
        self.p2 = p2  
    def suma(self) -> int:  
        return self.p1 + self.p2  
o1 = Clase(21, 122)
```

o1.suma()

la función `__init__` tiene dos parámetros marcados como enteros y la función `suma` tiene un retorno entero.

Los editores van a marcar aquello que considere que infiere una incompatibilidad de tipos.

Por otro lado, el **intellisense** de los editores va a ayudar al programador ya que se va a describir el tipo de los parámetros.

3.6 Programación defensiva

Cuando desarrollamos una función, debemos de validar que los parámetros se ajustan en tipo y que están dentro de un rango adecuado.

Si no son del tipo adecuado habrá que levantar una excepción `TypeError` y si el valor no es adecuado levantar la excepción `ValueError`.

A veces los parámetros son correctos en tipo y valor, pero la combinación de ellos no es posible. En este caso lanzaremos la excepción propia.

Cuando se produce un error, el usuario en producción nunca debe de recibir el stack trace del error. El usuario no sabría que hacer con esta información y por otro lado la información suministrada por el stack trace daría información que podría ser usada para atacar al sistema.

La comunicación al usuario, debe de ser del estilo:

“En estos momentos el sistema no funciona, inténtelo pasados unos minutos.”

Pero cuando se produce un error en producción debemos enterarnos los primeros, incluso antes que el usuario. Para ello el sistema, en producción, almacenará el error y el contexto asociados en un servicio, como `sentry`. `Sentry` podrá comunicar, vía notificación, con el servicio de mantenimiento correctivo, con el fin de solucionar el problema.

En el ambiente de desarrollo, las herramientas de desarrollo disponen de modo debug que estará activado y se dispone de ejecución paso a paso y funcionalidad para poder depurar el software.

3.7 Excepciones

Las excepciones son una parte importante del código. Las excepciones se pueden producir por que la ejecución del código no pueda continuar o por que detectemos una situación imposible en nuestros datos.

Si por ejemplo nos encontramos que un importe no puede ser negativo en el contexto de una función, podríamos hacer

```
if importe < 0:  
    raise Exception("El importe no puede ser negativo")
```

`Raise` lanzará una excepción genérica que hará que se acabe la función.

Cuando ocurre una excepción, tenemos la obligación de hacer log de los datos necesarios para que el programador pueda depurar el código

Esto lo podríamos hacer como sigue

```
try:
    llamada_que_da_error(parametros)
except:
    logging.error('texto para reflejar el error en el log ')
    raise
```

En el trozo de código anterior, se captura la excepción con el bloque `except`, pero para hacer log del error. **Raise** vuelve a levantar el error producido en la **llamada_que_da_error**

Hay ocasiones en las que se puede capturar el error y arreglarlo. Por ejemplo en el código siguiente:

```
try:
    fichero = open('fichero.txt', 'wb')
except:
    fichero = open('fichero.txt', 'wb')
# operaciones con fichero
```

En este caso, el programa abre un fichero y si no existe se crea. Es un caso en el que la excepción es capturada y se le da una solución.

Una **Exception** es un objeto y como tal admite herencia. Basándonos en esto, podemos definir excepciones custom creando clases derivadas de `Exception`. La siguiente es una custom excepción.

```
class TransaccionError(Exception):
    def __init__(self, message, errors):
        # Inicializo la clase base Exception con message
        super().__init__(message)
        # Se almacena las variables adicionales
        self.errors = errors
```

`self.errors` es una variable que pertenece a `TransaccionError`

Para levantar la excepción `TransaccionError` podríamos hacer

```
raise TransaccionError('No se pueden completar la transaccion',
    ['importe desconocido', 'cantidad menor que cero'])
```

A veces se produce un error en una función que tiene abierto un recurso software como puede ser una conexión a la base de datos. Es necesario liberar la conexión aunque la función produzca una excepción. Para eso disponemos del bloque finally.

try:

```
    db = open_database(URL_CONEXION)
```

```
    #operaciones
```

except:

```
    log
```

finally:

```
    close(db)
```

El bloque finally se va a ejecutar siempre, tanto si falla como sino. Esto asegura que la conexión a la base de datos será cerrada.

Dentro del desarrollo de aplicaciones hay que usar de manera adecuada las excepciones, cuidando en todo momento que cuando se produce un error hacer el log correspondiente para poder depurar el programa.

También hay que generar nuestra propia jerarquía de excepciones que permitan almacenar datos relevantes de nuestros módulos para así poder depurar.

Hay que pensar también que las excepciones son una forma de comunicar situaciones especiales. Así cuando nos encontramos validando parámetros en una función o cuando analicemos consistencia de los datos para realizar una transacción y no se cumpla lanzaremos una excepción, preferentemente personalizada, a la que llevemos los datos de la transacción y la razón de la inconsistencia.

3.8 Modularidad.

Uno de los objetivos cuando se realiza software es el encapsular funcionalidad, de tal forma que esta sea fácil de usar.

Python permite la creación de módulos, que podremos importar en nuestro código. Estos módulos pueden estar compuestos de funciones. Tendremos a nuestra disposición muchas librerías de funciones.

Por otro lado, Python es un lenguaje en el que se dispone de la orientación a objetos. Desde este punto de vista podemos usar módulos existentes que contiene clases y por lo tanto podemos instanciar objetos y también podemos crear nuestra propia jerarquía de clases que represente nuestra aplicación.

Como recomendación, usaremos clases y objetos, ya que existen metodologías orientadas a la creación de modelos de clases que pueden luego implementarse en el lenguaje Python.

4 Organización del código Django

DJANGO es un framework escrito en PYTHON. Django es un open source cuyo origen se remonta al 2005. En el 2008 se crea la Django Software Foundation, que se encarga de la evolución del proyecto.

El framework a día de hoy ha evolucionado mucho y es un marco de trabajo que proporciona una alta capacidad de producción de software, potenciando la reutilización, haciendo fácil y uniforme la conexión a diferentes motores de base de datos con su ORM (Object Relationship Management).

A finales del 2019 se espera la versión 3.0 del framework que usa las características que se centra en que el framework sea capaz de funcionar de manera asíncrona, con librerías como Daphne o Uvicorn.

4.1 La estructura de una aplicación Django

La figura siguiente describe el funcionamiento de una aplicación Django

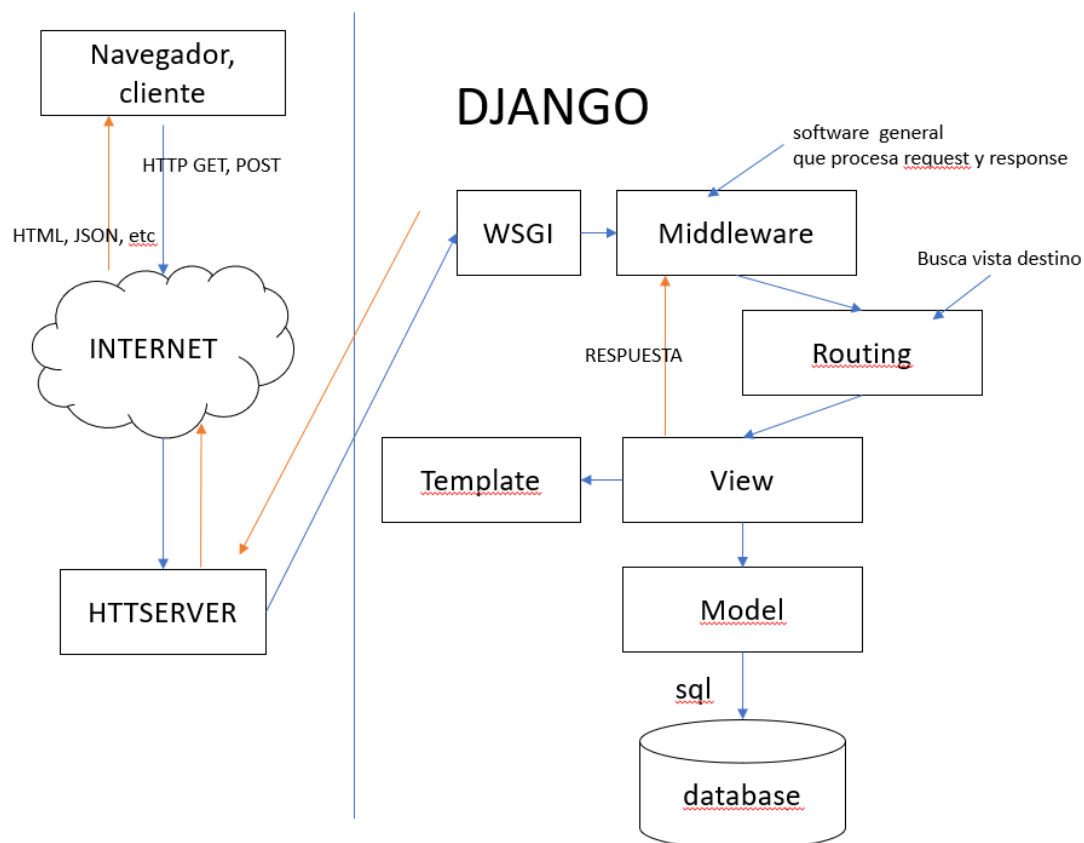


Ilustración 3 Estructura aplicación django

<p>Navegador, Cliente</p>	<p>Es el agente que solicita datos al servidor. En la mayoría de los casos será un navegador, pero puede ser otro agente. Se comunica con el servidor por medio del protocolo http</p>
---------------------------	--

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Internet	Red de comunicación que permite alcanzar el servidor. Será una red tcp/ip.
HttpServer	Es el servidor que escucha las peticiones. Entre los más conocidos están nginx, apache, iis
WSGI	Modelo de interacción entre servidores web y aplicaciones.
Middleware	Software python dentro de django, definido en settings y que permite alterar la request y/o la response.
Routing	Módulo de django que usa las definiciones en el fichero urls.py, para seleccionar la vista
View	Código python que recibe la request y la procesa y usando las templates genera la respuesta
Model	La vista usa las clases que representan el modelo de datos. Estas clases del modelo accederán a la base de datos vía sql
Template	En conjunción con la vista, describe la salida, que será principalmente html. Usará datos proporcionados por la vista, que vendrán de la base de datos

Tabla 4 Tabla de componentes Django

En la siguiente figura describimos los componentes que se fabricarán en el proceso del desarrollo

COMPONENTES DE DJANGO

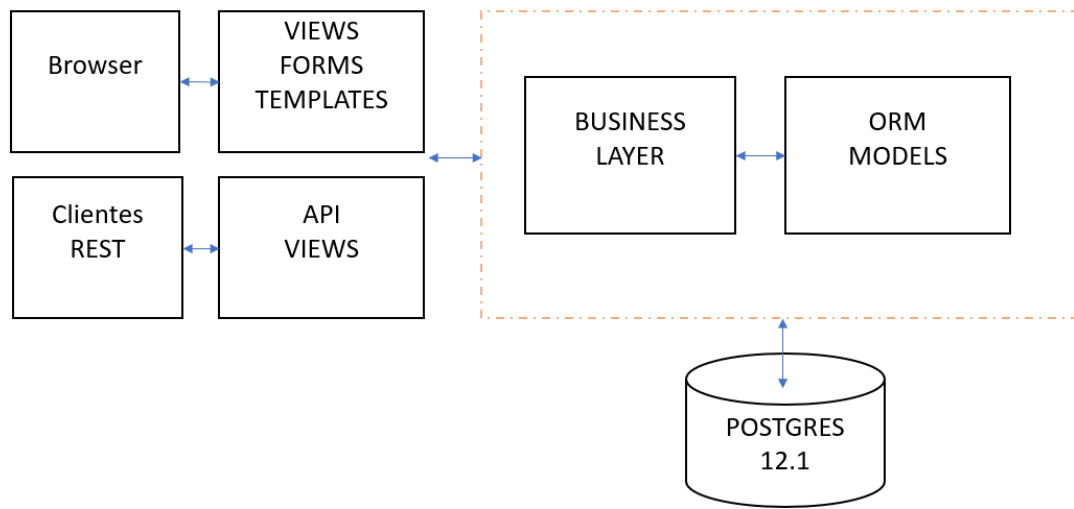


Tabla 5 Componentes Django

Descripción de los distintos componentes Django para la construcción de aplicaciones Django

ORM	Permite la definición del modelo de datos. Es independiente de la base de datos a usar. Puede trabajar con Mysql, Postgres. SQLite y en la versión 3.0 con MariaDB
VIEWS	Permite definir vistas, que representan lo que ve el usuario. Unido a las Views están las templates que son básicamente el HTML que se representará en el navegador
FORMS	Los formularios son una parte importante en el desarrollo web. Tienen relación con el modelo de datos y con la presentación de formularios html que se presentarán al usuario
BL	(Business Layer) es la capa de software dentro de Django que contiene el código de las reglas de negocio. Tanto el api como las views deben de usar código ubicado en esta capa.
API	Si nuestro sistema necesita un API, podemos realizarlo con un módulo específico llamado Django Rest Framework, que nos ayuda en todas las tareas de serialización / deserialización

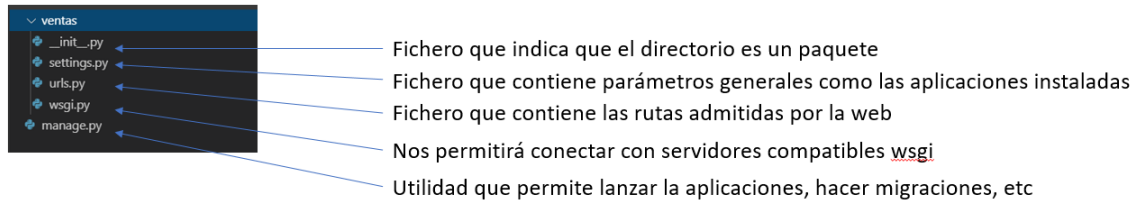
Tabla 6 Componentes de django

4.2 Creación de una aplicación web django

Vamos a crear un proyecto llamado ventas con el comando

```
django-admin startproject ventas
```

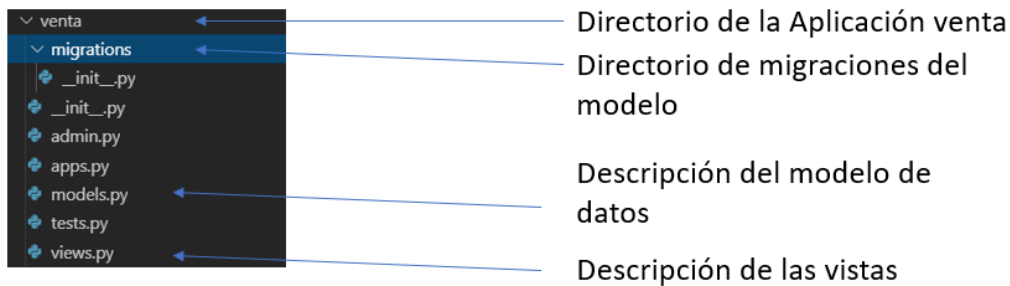
Este comando crea un directorio llamado ventas con el siguiente contenido



Un proyecto django tiene un conjunto de aplicaciones. Para crear una aplicación hacemos

```
>python manage.py startapp venta
```

Esto crea el el directorio venta con la siguiente estructura



Tenemos que añadir la aplicación al proyecto. Para ello, en el fichero settings.py añadimos en el apartado INSTALLED_APPS la clase **venta.apps.VentaConfig**.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'venta.apps.VentaConfig',  
]
```

Hemos acabado el setup del proyecto y de la aplicación.

Para lanzar la aplicación hacemos

```
>python manage.py runserver 0.0.0.0:8081
```

Esto lanza el servidor web y le permite escuchar en cualquier ip del servidor y el puerto **8081**

Si nos vamos a un navegador y vamos a localhost:8081 tenemos

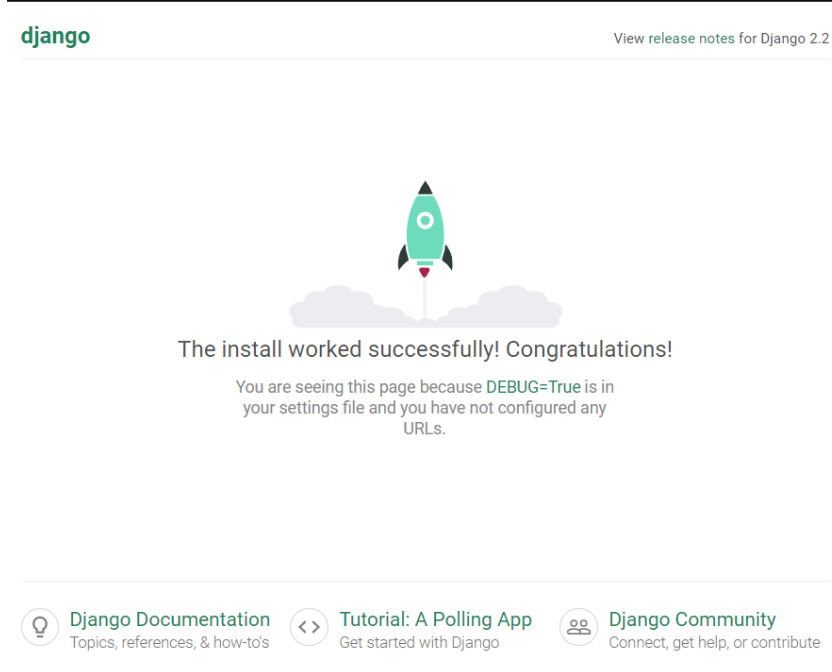


Ilustración 4 Página inicio aplicación django

Esto nos indica que lo hemos instalado y configurado correctamente.

NOTA: Si nuestra ip es 9.56.6.85 y que queremos ejecutar la dirección <http://9.56.6.85> debemos de declarar esta dirección en settings.py en el array **ALLOWED_HOSTS**. Si no lo hacemos el servidor nos devuelve

DisallowedHost at /

Invalid HTTP_HOST header: '9.56.6.85:8081'. You may need to add '9.56.6.85' to ALLOWED_HOSTS.

```
Request Method: GET
Request URL: http://9.56.6.85:8081/
Django Version: 2.2.7
Exception Type: DisallowedHost
Exception Value: Invalid HTTP_HOST header: '9.56.6.85:8081'. You may need to add '9.56.6.85' to ALLOWED_HOSTS.
Exception Location: C:\Users\jviejo\AppData\Local\Continuum\anaconda3\envs\p37\lib\site-packages\django\http\request.py in get_host, line 111
Python Executable: C:\Users\jviejo\AppData\Local\Continuum\anaconda3\envs\p37\python.exe
Python Version: 3.7.5
Python Path: ['C:\\Users\\jviejo\\djangoprojects\\ventas',
'C:\\Users\\jviejo\\AppData\\Local\\Continuum\\anaconda3\\envs\\p37\\python37.zip',
'C:\\Users\\jviejo\\AppData\\Local\\Continuum\\anaconda3\\envs\\p37\\DLLs',
'C:\\Users\\jviejo\\AppData\\Local\\Continuum\\anaconda3\\envs\\p37\\lib',
'C:\\Users\\jviejo\\AppData\\Local\\Continuum\\anaconda3\\envs\\p37',
'C:\\Users\\jviejo\\AppData\\Roaming\\Python\\Python37\\site-packages',
'C:\\Users\\jviejo\\AppData\\Local\\Continuum\\anaconda3\\envs\\p37\\lib\\site-packages']
Server time: Tue, 5 Nov 2019 08:33:37 +0000
```

Ilustración 5 Error de ip no permitida

4.3 Los modelos

Es una de las partes más importante del framework de Django. El ORM de Django nos permite definir a través de código python las tablas, relaciones, índices. Una vez definidas las clases éstas son transformadas en código sql que permite la creación de las tablas y relaciones en la base de datos.

El ORM permite el cambio del modelo, pudiendo añadir, modificar eliminar campos. Estas definiciones, realizadas en código python dentro del framework producirán los cambios en el modelo de la base de datos elegida. Se crearán campos, se cambiará su definición o se quitarán campos. También pueden crearse relaciones o quitarse. A la operación de sincronizar el modelo en python con el modelo físico de la base de datos se le conoce con el nombre de migración.

En la siguiente figura se aprecia el código Django ORM, una clase python, y el correspondiente SQL en el que se traduce.

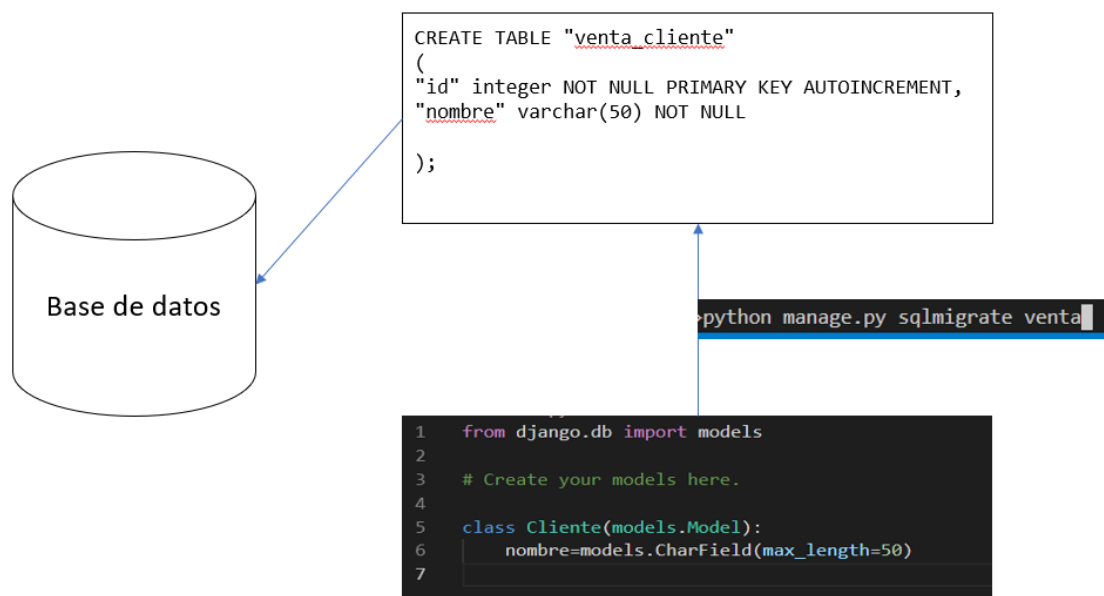


Ilustración 6 Relación modelo python y sql

Hay que destacar que para el programador será más fácil mantener el código python que el modelo de base de datos con SQL. El sistema de migración sincroniza el Django model con las tablas en SQL.

Con la orden

```
python manage.py makemigrations
```

se traducirá el código de las clases del modelo en sentencias sql. Nótese que el mecanismo de migración añade un id a la tabla declarado como primary key AUTOINCREMENT.

Lo que en realidad hace “makemigrations” es crear un archivo dentro de migrations, en este caso 0001_initial.py, que contiene una clase Python con el modelo. Para que este modelo se cree en la base de datos haremos

```
python manage.py migrate
```

El comando crea en la base de datos la tabla Cliente. Pero, ¿en qué base de datos crea la tabla?. Pues bien, en el fichero **settings.py** tenemos la definición siguiente

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

que indica que el gestor de base de datos es **sqlite3** y que la base de datos estará en el fichero **db.sqlite3**. **BASE_DIR** representa el directorio del proyecto y está definida en el mismo fichero settings.py

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

Si queremos usar otra base de datos tendríamos que cambiar la definición **DATABASES**. Por ejemplo

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': "aplicacion",
        'USER': "usuario_postgres",
        'PASSWORD': "password",
        'HOST': "host_donde_esta_el_postgres",
        'PORT': 5432,
    }
}
```

usa el motor de base de datos postgres. Se configura también el usuario/pwd y el host/port donde está el servidor de base de datos.

Las bases de datos soportadas por el framework son: MySQL, Postgres, Oracle, sqlite. Terceras partes han desarrollado drivers para IBM DB2, Microsoft SQL Server, Firebird, ODBC.

4.3.1 Los tipos de datos

Cuando definimos el modelo podemos elegir entre los tipos siguientes:

```
__all__ = [
    'AutoField', 'BLANK_CHOICE_DASH', 'BigAutoField', 'BigIntegerField',
```

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

```
'BinaryField', 'BooleanField', 'CharField',  
'CommaSeparatedIntegerField',  
'DateField', 'DateTimeField', 'DecimalField', 'DurationField',  
'EmailField', 'Empty', 'Field', 'FieldDoesNotExist', 'FilePathField',  
'FloatField', 'GenericIPAddressField', 'IPAddressField',  
'IntegerField',  
'NOT_PROVIDED', 'NullBooleanField', 'PositiveIntegerField',  
'PositiveSmallIntegerField', 'SlugField', 'SmallIntegerField',  
'TextField',  
'TimeField', 'URLField', 'UUIDField',  
  
]
```

Cada tipo de campo , se mapea con un tipo de columna de la base de datos. Así tenemos campos de fecha, numéricos, binarios, booleans, etc.

Aparte de estos tipos que vienen con el framework, hay empresas que crear otros tipos de campos, ampliando así la riqueza de las definiciones.

Por otro lado, las bases de datos relacionales tiene características especiales que quizás queremos usar. Por ejemplo postgres tiene un tipo de datos range (tipo daterange). Para poder definir un tipo rango en nuestro modelo de django usaremos el paquete **django.contrib.postgres**, que trae definiciones específicas para rangos de enteros, fechas, etc.

Hay que tener en cuenta que usar tipos de datos específicos de un motor de base de datos hace que la aplicación no sea transportable. Si lo que queremos es usar todas las características de la base de datos es interesante usar las extensiones.

Si nos decidimos usar postgres como base de datos, el paquete **django.contrib.postgres**, contiene los siguiente tipos de campos:

ArrayField	Podemos almacenar en una columna una lista de datos
HStoreField	Un campo para almacenar clave/valor
JSONField	Una campo para almacenar datos codificados en formato JSON
IntegerRangeField	Almacena un rango de enteros (enteros de 4 bytes)
BigIntegerRangeField	Almacena un rango de enteros (enteros de 8 bytes)
DecimalRangeField	Almacena un rango. El tipo postgres es numrange
FloatRangeField	Almacena un rango de float
DateTimeRangeField	Basado en el tipo tstzrange de postgres
DateRangeField	Basado en el tipo daterange de postgres

Tabla 7Tipos de campos

Aparte del mapeo de campos a campos de la base de datos, las clases implementan operaciones que permiten operadores especiales que se traducirán con operadores especiales sql definidas en base de datos.

4.3.2 Operaciones con el modelo

Vamos a ver como realizamos operaciones sobre el modelo. Para ello vamos a usar el comando siguiente

```
python manage.py shell
```

Nos aparece

```
(p37) C:\Users\dviejo\djangoprojects\ventas>python manage.py shell
Python 3.7.5 (default, Oct 31 2019, 15:18:51) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Esta consola interactiva nos va a permitir ejecutar sentencias python.

```
#nos permite importar todas las clases definidas en el paquete
venta.models.
#recordamos que venta es una aplicación dentro del proyecto
django
>>> from venta.models import *
# la siguiente instrucción recupera de la base de datos todos
los objetos de la tabla (modelo) Cliente
>>> q = Cliente.objects.all()
# la consulta devuelve una lista de objetos
>>> <QuerySet []>
# podemos ver la sentencia sql en que se traduce la expresión
python anterior
>>> print(q.query)
# esta es la query que lanza a la base de datos
SELECT "venta_cliente"."id",
"venta_cliente"."nombre", "venta_cliente"."telefonos" FROM
"venta_cliente"
>>>
```

Ahora vamos a ver como insertamos información a través de las clases de python

El modelo es:

```
from django.db import models
from django.contrib.postgres.fields import *
# Create your models here.

class
  Cliente(models.Model):
    nombre=models.CharField(max_length=50)
    telefonos= array.ArrayField(
      models.CharField(max_length=15), size=8, null=True
    )
```

Se trata de una tabla que tiene el nombre y un array de teléfonos. Estamos usando el postgres con un tipo de campo llamado ArrayField que permite almacenar una lista

```
cli = Cliente.objects.create(nombre='VIEJO POMATA DAVID',  
telefonos=['555121212', '555121212'])
```

Exploramos el objeto devuelto

```
>>> cli  
<Cliente: Cliente object (2)>  
>>> cli.id  
2  
>>> cli.nombre  
'VIEJO POMATA DAVID'  
>>> cli.telefonos  
['555121212', '555121212']  
>>>
```

Si queremos borrar el registro, esto lo podemos hacer con

```
cli.delete()
```

Si queremos añadir un teléfono a la lista de teléfonos de la persona, tendremos que actualizar el registro

```
cli.telefonos.append('555333444')  
cli.save()
```

Como teléfonos es un array, con append añadimos un teléfono mas. Con save grabamos a la base de datos.

Si queremos acceder a un cliente por la clave, que es un autonumérico haremos

```
cli = Clientes.objects.get(pk=3)
```

o

```
cli = Cliente.objects.get(id=3)
```

get nos proporciona un registro.

Si queremos aquellos clientes sin telefono

```
clientes = Cliente.objects.filter(telefonos__isnull=True)
```

El método filter devuelve 0 o más registros. Como parámetros del “filter” tenemos condiciones. En este caso **telefonos__isnull=True**. Nótese que tenemos telefono unico con isnull por __. Lo que ocurre es que se aplica la operación isnull sobre el campos teléfonos. y el resultado se compara con True.

El ORM traduce el filtro a

```
SELECT "venta_cliente"."id", "venta_cliente"."nombre",
```



```
"venta_cliente"."telefonos" FROM "venta_cliente" WHERE
"venta_cliente"."telefonos" IS NULL
```

4.4 La lógica de negocio

Entendemos por lógica de negocio la que representa las funciones que debe de realizar el sistema que estamos desarrollando.

La lógica de negocio realizará cambios en el modelo en varias entidades a través del ORM.

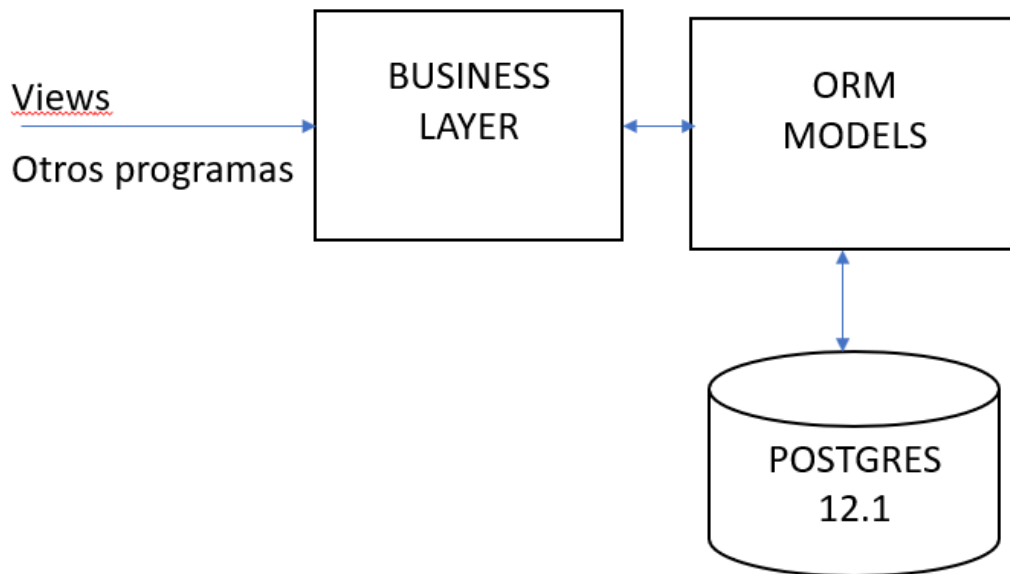


Ilustración 7 Esquema interacción modelo de negocio

La lógica de negocio será usada en Django por vistas, pero podría ser usado por otros programas python.

La capa de negocio usará los modelos para cambiar el estado de la base de datos.

En el ORM de Django cada tabla de la base de datos está representado por una clase Python. Esta clase python puede ser extendida con funciones que realicen algunas operaciones sobre atributos del modelo e incluso sobre otros modelos.

Por ejemplo:

```
class VentaLinea(models.Model):
    venta = models.ForeignKey(Venta, on_delete=models.CASCADE)
    producto = models.ForeignKey(Producto, on_delete=models.CASCADE)
    cantidad = models.DecimalField(decimal_places=2, max_digits=10)
    precio = models.DecimalField(decimal_places=2, max_digits=10)
    descuento = models.DecimalField(decimal_places=2, max_digits=4)
    iva = models.DecimalField(decimal_places=2, max_digits=4)
    def importe_bruto(self):
        return round(self.cantidad * self.precio, 2)
    def importe_descuento(self):
```

```
        return round(self.importe_bruto * self.descuento, 2)
def importe_con_descuento(self):
    return (self.importe_bruto - self.importe_descuento)
def importe_iva(self):
    return round(self.importe_con_descuento * self.iva, 2)
def importe_total(self):
    return self.importe_con_descuento - self.importe_iva
```

En este modelo, además de los atributos definidos, tenemos métodos que permite obtener datos calculados a partir de los atributos del modelo.

Hay varias tendencias en cuanto a poner métodos adicionales en los modelos. Hay una tendencia que defiende que cuanto más lógica se introduzca en los modelos será mejor ya que se reutilizará más.

Por otro lado, hay otra tendencia que propone solo meter métodos que solo afecten a atributos de la entidad.

La recomendación en esta arquitectura es extender las clases que representan la tabla con funciones basadas en atributos de la clase.

4.5 Las vistas y los formularios

En una aplicación web el usuario interactúa con la página lo cual produce una petición http lo cual es recogida el servidor. En el caso del Django es recogida por el componente view.

Las vistas permiten procesar peticiones http como GET, POST. La vista conecta con la TEMPLATE. Las template contienen el HTML que va a ser la base para formar el HTML de respuesta.

Ejemplo de vista con template:

```
class EjemploTemplateView(TemplateView):
    template_name="template.html"

    def get_context_data(self, **kwargs):
        context = super(EjemploTemplateView, self)
            .get_context_data(**kwargs)
        context['q'] = self.request.GET
        return context
```

El ejemplo que hemos puesto está basado en el uso de Class-based view (4).

Django permite el uso de funciones como view (5) aunque no se recomienda el uso en la arquitectura que definimos.

```
from django.http import HttpResponse

import datetime

def current_datetime(request):
```

```
now = datetime.datetime.now()

html = "<html><body>It is now %s.</body></html>" % now

return HttpResponse(html)
```

En el ejemplo anterior tenemos una función llamada `current_datetime` que devuelve un html a través del objeto `HttpResponse`. No se usa ninguna template.

Sigue funcionando en versiones actuales pero está desaconsejado.

4.5.1 Relación entre urls y vistas

Como sabemos cuando el usuario hace click en un link, se ejecuta una petición http en la que viaja una url con parámetros. Si se envían datos de un formulario se ejecuta una petición get o post según esté definido y va una url con parámetros o con un body

Ejemplo de POST con body

```
POST /formularios/cliente HTTP/1.1
```

```
Host: servidor.com
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 13
```

```
nombre=DAVID&apellidos=VIEJO POMATA
```

En el ejemplo anterior el navegador hace un POST a la url `/formularios/cliente` y se le pasan los datos nombre y apellidos.

Cuando este contenido llegue a Django será analizado y usando la tabla de relación urls con vistas se obtendrá la la vista que toca

Este fichero se llama `urls.py`

```
urlpatterns = [
    path('/formularios/cliente', ClienteView.as_view(), name='chat'),
]
```

Con el contenido anterior en el fichero `urls.py` el Django nos da que es la vista **ClienteView** la que recibirá los datos de la petición.

4.5.2 Tipos de Class-based views

Dos de las templates más básicas son

TemplateView	Nos sirve para renderizar html a partir de una template
RedirectView	Nos sirve para realizar un redirect.

Si queremos usar una template a la pasemos un conjunto de información podemos usar la template view

```
from django.views.generic.base import TemplateView
```

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

```
class ArtículoPageView(TemplateView):  
    template_name = "articulo.html"  
  
    def get_context_data(self, **kwargs):  
        context = super().get_context_data(**kwargs)  
  
        context['lista'] = <accedemos a la lógica de negocio para  
obtener datos>  
  
        return context
```

En **context** introducimos los objetos que necesitamos en la vista.

```
<table class="table">  
<thead  
    <tr>  
        <th>Cabecera ..</th>  
        <th>Cabecera ..</th>  
    <tr>  
</thead>  
<tbody>  
{% for item in lista %}  
    <tr>  
        <td>{{ item.atributo }}  
    </tr>  
{% endfor %}  
</tbody>  
</table>
```

El lenguaje de la vista es simple. Podemos usar las estructuras básicas como la condicional y la repetitiva. Esto lo hacemos con tags

<pre>{% for object i in objects %} contenido del bucle {% endfor %}</pre>	El contenido puede ser código html u otros tags. Cada tag tiene un inicio y un cierre
<pre>{% if condicion %} Contenido si se cumple la condición {% else %}</pre>	Estructura condicional en una template

<pre>Contenido del else {% endif %}</pre>	
---	--

Si lo que se quiere es usar alguna variable almacenada en el context, eso se hace usando variables que se representan en la vista como

```
{{ variable.atributo }}
```

Se usan los símbolos `{{}}` para definir una variable en la plantilla.

Se puede alterar la representación de las variables con los **filtros**. Un ejemplo es

```
{{ fecha_inicio |date:"d/m/Y" }}
```

Lo anterior permite formatear una objeto de tipo fecha de una manera específica. El símbolo `|` hace que el dato fecha se pase a un filtro llamado **date**, con el parámetro **d/m/Y**.

Es posible realizar filtros especiales para nuestros datos. Imaginemos que queremos hacer un filtro para que se presente siempre en mayúsculas el dato.

```
from django.template.defaultfilters import stringfilter
register = template.Library()
@register.filter
@stringfilter
def mayusculas(value):
    return value.lower()
```

Se han usado atributos para definir la función `mayusculas` como filtro y que admite un string. El decorador `@register.filter` permite registrar la función para ser usada como filtro. El decorador `@stringfilter` define que el parámetros será una cadena.

En un desarrollo grande conviene hacer custom filter con el fin de personalizar la presentación de ciertos objetos y también con el objetivo de reutilizar y unificar estas presentaciones.

4.5.3 Los formularios

Los formularios son una parte esencial de las aplicaciones web ya que permiten al usuario introducir información.

En cuanto a la programación web, suelen ser un de las partes más complejas del sistema. Django suministra una serie de `ClassViews` orientadas a trabajar con formularios.

Django dispone de vistas para poder hacer mantenimiento de entidades, creación, actualización, borrado, lista, detalle.

CreateView	Para crear un registro
DetailView	Para ver los detalles
DeleteView	Para pedir confirmación al borrado de un registro
ListView	Para ver la lista de registros

Las siguientes vistas sirven para mantener los datos de una tabla

```
class ClienteCreateView(CreateView):
    model = Cliente
    fields = ['identificacion', 'nombre']
    def get_success_url(self, **kwargs):
        return reverse('list_clientes')
```

Html para el formulario para introducir registros

```
<form method="post">
    {% csrf_token %}
    <table>
        {{form}}
    </table>
    <p><input type="submit" value="Save"></p>
</form>
```

El tag **csrf_token** unido al middleware CsrfViewMiddleware permite proteger el sistema frente al ataque Cross Site Request Forgery.

Modificación de un registro

```
class ClienteUpdateView(UpdateView):
    model = Cliente
    fields = ['identificacion', 'nombre']
    def get_success_url(self, **kwargs):
        return reverse('list_clientes')
```

Para ver los detalles del registro

```
class ClienteDetailView(DetailView):
    model = Cliente
    def get_success_url(self, **kwargs):
        return reverse('list_clientes')
```

```
<p>{{ object.identificacion }}</p>
<p>{{ object.nombre }}</p>
<a href="{% url 'list_clientes' %}">Lista</a>
```

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Para borrar un cliente tenemos la vista DeleteView

```
class ClienteDeleteView(DeleteView):
    model = Cliente
    def get_success_url(self, **kwargs):
        return reverse('list_clientes')
```

Html del formulario para pedir confirmación

```
<form method="post">{% csrf_token %}
    <p>Estas seguro que quieres borrar el registro "{{ object }}"?</p>
    <input type="submit" value="Confirm">
</form>
```

Para obtener la lista de clientes

```
class ClienteListView(ListView):
    model = Cliente
```

Template para ver clientes

```
<h1>Clientes</h1>
<ul>
{% for cliente in object_list %}
    <li>
        <a href="{% url 'cliente_detail' cliente.id %}">Detalle</a>
        <a href="{% url 'cliente_update' cliente.id %}">Edit</a>
        <a href="{% url 'cliente_delete' cliente.id %}">Borrar </a>
        {{ cliente.identificacion }} - {{ cliente.nombre }}</li>
{% empty %}
    <li>No clientes todavia</li>
{% endfor %}
```

Si nos fijamos en las clases anteriores vemos que no hay ninguna instrucción para crear, borrar, leer, actualizar registros del modelo. Las clases tienen un atributo llamado `model` que representa el modelo definido, que estará asociado a una tabla de la base de datos a través del `orm` de django.

Tampoco hemos referenciado cual es la template que usan las vistas. Si no decimos nada los nombres son

- `Cliente_detail.html` para la vista `ClienteDetailView`
- `Cliente_list.html` para para la vista `ClienteListView`
- `Cliente_confirm_delete` para la vista `ClienteDeleteView`
- `Cliente_form` para las vistas `ClienteCreateView`, `ClienteUpdateView`

Podemos personalizar la template que usara la clase con

```
class ClienteListView(ListView):
    model = Cliente
    template_name = "lista_de_clientes.html"
```

Tampoco estamos indicando cómo deben aparecer los formularios. Si queremos personalizar los formularios, debemos de usar clases que indican como son los formularios. Estas clases son FormView o ModelView. FormView define un formulario no asociado a ningún modelo y ModelView define un formulario basado en un modelo

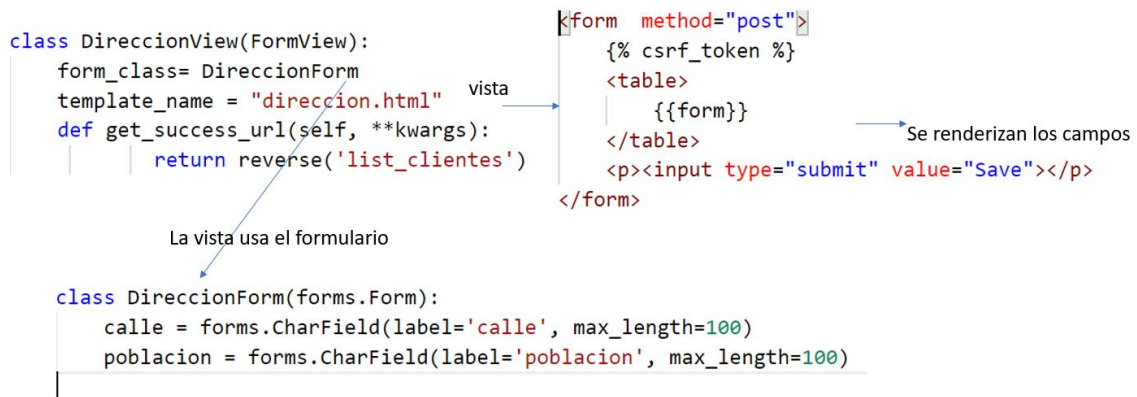


Ilustración 8 Uso de formularios en django

En la figura anterior vemos la vista, que hereda de FormView (no está unida a ningún modelo). La vista usa la definición de un formulario a través de la clase DireccionForm (define dos campos). Por otro lado, tenemos la template que define el html, donde `{{form}}` representa el html del form definido.

4.5.4 Formularios basados en modelos

En el apartado anterior hemos visto el FormView que permitía definir un formulario sin atarlo a un modelo.

Es posible que nos interese, sobre todo en aplicaciones data driven, formularios basados en modelos. En este caso el sistema nos proporciona los elementos de formulario adecuados.

```
from django.forms import ModelForm

class AuthorForm(ModelForm):

    class Meta:

        model = Cliente

        fields = '__all__'
```

Con la definición anterior, estamos definiendo un formulario basado en el modelo Cliente y estamos poniendo todos los campos. Si quisiéramos poner algunos campos deberíamos de definir fields con un array de nombres de campos.

Con la definición anterior django no tenemos control sobre la presentación que hace django del formulario. Si queremos personalizarlo debemos de usar lo siguiente


```

from django.forms import ModelForm, Textarea

from myapp.models import Cliente

class ClienteForm(ModelForm):

    class Meta:

        model = Cliente

        fields = ('dni', 'nombre', 'asunto')

        widgets = {

            'asunto': Textarea(attrs={'cols': 100, 'rows': 10}),

        }

```

En el caso anterior, hemos personalizado el asunto para que aparezca como un textarea con 10 filas de 100 columnas.

Para usar el formulario definido anteriormente en una vista hacemos lo siguiente:

```

from myapp.forms import ClienteForm

class ClienteCreateView(ModelForm):

    template_name= 'create_cliente.html'

    form_class=ClienteForm

```

Recordamos que la vista es el elemento que responde a una petición desde el navegador.

4.6 Las validaciones en los formularios.

Un problema importante en los formularios es validar los datos son válidos y que no almacenamos datos incompletos en la base de datos.

Los forms nos permite definir el tipo de campos y además definir validaciones que se pasarán a los datos introducidos por el usuario. Los forms permiten pasar validaciones a campos individuales y validaciones a grupos de campos.

Los forms disponen de una función llamada clean, que se ejecuta cuando el usuario hace post en un formulario.

```

def clean(self):
    cleaned_data = super(FormularioForm, self).clean()
    if cleaned_data["campo"] == None:
        raise forms.ValidationError(
            message=_("El Mensaje de error"),
            code="ERR_001_004"
        )
    return cleaned_data

```

Esta función, en el caso de que el dato de campo sea None, levantará una excepción en la que podemos poner una descripción del error. La descripción del error debe de ser lo suficientemente explicativa para el usuario.

Podemos también definir una función clean para un campo como la siguiente

```
def clean_requerimiento(self):
    requerimiento = self.cleaned_data["requerimiento"]
    if not Eni.objects.filter(codigo=requerimiento).count():
        raise forms.ValidationError
            _("El requerimiento %s no existe") % (requerimiento,))
    return requerimiento
```

En este caso, tenemos el campo requerimiento. Con el accedemos al modelo Eni y si se cumple la condición levantamos la excepción.

Hemos visto el uso del clean para validar, pero lo podríamos usar para transformar la entrada recibida en el POST.

```
def clean_email(self):
    """
        Asegura que el mail se pasa a minúsculas
        aunque el usuario haya puesto letras mayúsculas y
        minúsculas.
    """
    return self.cleaned_data['email'].lower()
```

En django forms tenemos el concepto de errores asociados a una campo y errores no asociados a ningún campo, que son errores asociados al formulario.

4.7 Las templates

Las template sirven en Django para generar la salida que irá al navegador. Son muy importantes en una aplicación web al ser el componente que define lo que verá el usuario. Estará cargada de elementos de diseño visual y de user experience que haga que el producto sea atractivo.

Una de las decisiones que hemos de tomar cuando comencemos un sistema en Django es decidir qué componentes visuales externos a Django vamos a usar y que framework css vamos a usar.

El framework más usado en css es el Bootstrap. El framework css tiene importancia no solo por lo que nos permite hacer, sino por los componentes que usan Bootstrap y que nos interesa usar.

Unido a Bootstrap está jquery que es la librería de javascript más usada en la actualidad. Es un complemento bueno para realizar interacciones sin el refresco de página.

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Existen frameworks completos de cliente como Angular, React, Vue que nos facilitan la creación de páginas en cliente. Se basan en que se carga toda la aplicación nada más arrancar la página y luego esta accede al servidor vía llamada REST.

Si la aplicación necesita una interactividad muy alta, podríamos elegir algunos de estos frameworks. También es posible usar un modelo mixto, páginas realizadas en el servidor con templates y alguna otra con, por ejemplo, Vue. Vue es de los frameworks que menos pesa y permita una alta interactividad.

En el sistema de templates de Django se extienden unas de otras

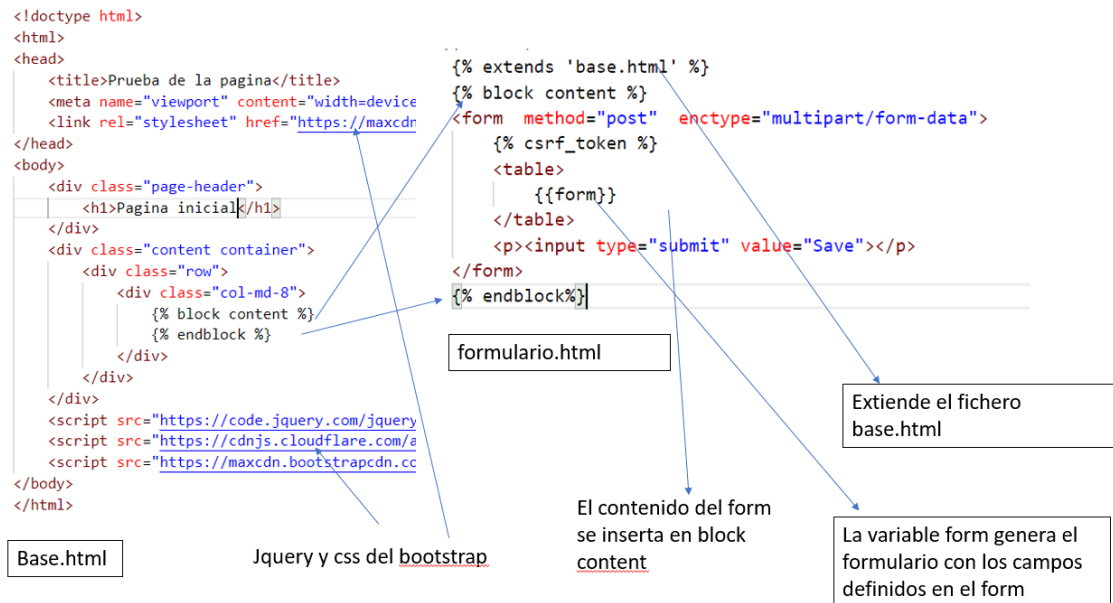


Ilustración 9 Sistema de templates de Django

En la ilustración anterior tenemos un archivo llamado base que contiene la plantilla general del HTML. A la derecha tenemos una plantilla llamada formulario que extiende de base.html. En la plantilla base.html hay un bloque llamado content, que en la plantilla formulario es sustituido por el form.

Una plantilla Django es un texto donde podemos poner unas marcas que indican al procesador de la plantilla lo que tiene que hacer.

Básicamente tiene variables, tags y filters. Las variables que van encerradas entre `{{}}` hacen que se sustituya el valor de la variable.

Los tags proporcionan lógica a las plantillas que permiten que haya contenido condicional con el tag "if" o contenido repetitivo con el tag "for". Es posible crear tags personalizadas con lógica personalizada.

```
{% if descuento > 0 %} Descuento Aplicado {% endif %}
```

Si queremos presentar en la página los headers recibidos:

```
{% for key, value in request.headers.items %}
  <p>{{key}}:{{ value }}</p>
{% endfor %}
```

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Request.headers.items representa un diccionario con los headers de la request. El tag for nos permite recorrer todos los elementos del diccionario. Por cada elemento se tiene su key y su value.

Salida parcial del código anterior

Content-Length:

Content-Type:text/plain

Host:localhost:8000

Connection:keep-alive

Cache-Control:max-age=0

Upgrade-Insecure-Requests:1

User-Agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36

Luego tenemos los filters que transforman el contenido de una variable.

Por ejemplo

```
{{'esto saldrá en mayúsculas por el filtro upper' | upper}}
```

El filtro upper pasa la salida a mayúsculas. Se con una variable seguida del símbolo | al que le sigue el nombre del filtro.

Salida del filtro

ESTO SALDRA EN MAYUSCULAS POR EL FILTRO

Podemos realizar tanto tags como filtros a medida de nuestras necesidades, para facilitar la presentación de los datos.

Otra funcionalidad de las templates es la de incluir una template en otro con el tag include

4.8 Apis con Django Rest Framework

En el ecosistema Django existen las vistas que unidas a las templates dan soporte al funcionamiento de una página web, pero si queremos hacer un api, porque nuestro sistema va a ser accesible por aplicaciones móviles, por ejemplo, podemos usar una librería no incluida en django que permite la gestión de vistas orientadas a recibir peticiones REST (6).

Las peticiones REST, que son peticiones http con una cierta sintaxis. Los clientes REST envían peticiones en las que suelen enviar información en formato json y recibir información en json.

Las operaciones REST más habituales son

GET	Leer una entidad o un conjunto de entidades
PUT	Actualizar una entidad
POST	Crear una entidad
DELETE	Borrar una entidad

Lo que nos hace la librería REST Framework es facilitar la interacción con clientes REST. Para ello dispone de los siguientes componentes:

- Serializadores
- ViewSet
- Routers

Los serializadores nos permite definir como se empaquetarán los datos en la respuesta.

Los viewset recibirán las peticiones GET, POST, DELETE, PUT que permitirán crear, actualizar, leer entidades.

Los routers nos permite añadir rutas que permitan acceder a entidades.

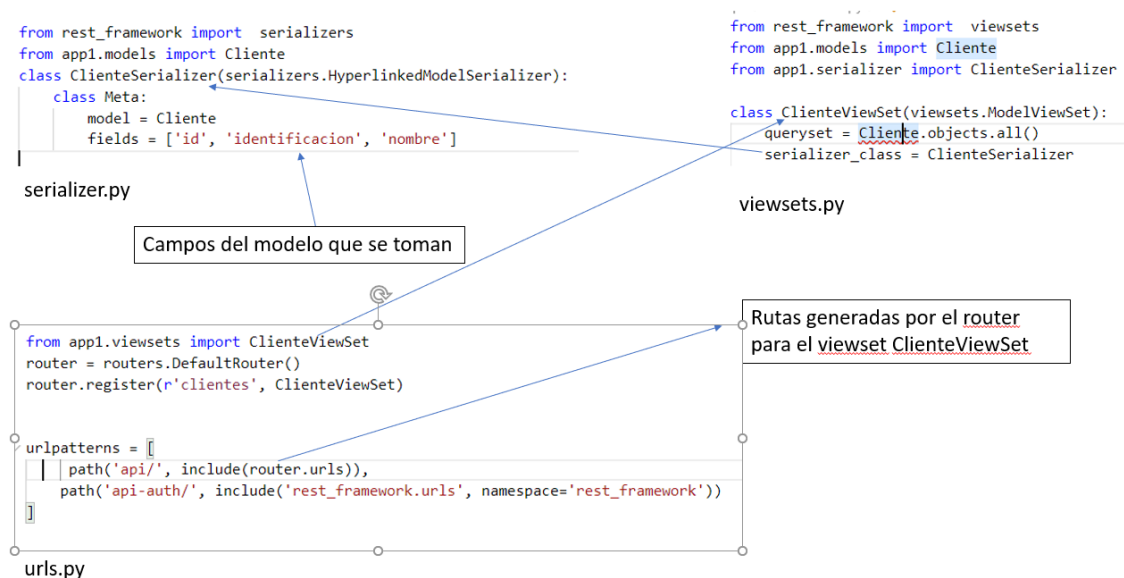


Ilustración 10 Funcionamiento de django rest framework

En la figura anterior tenemos el fichero `serializer.py` que serializa los elementos de la entidad `Cliente`. También tenemos la clase `ClienteViewSet` que hereda de la clase `ModelViewSet` y que tiene la lógica para responder al GET, POST, DELETE y PUT y por último en el fichero `urls.py` tenemos la definición del router en el cual registramos `ClienteViewSet`. Las url generadas por el router se añaden al arrays `urlpatterns` utilizado por Django para asociar peticiones con funciones o clases.

Esta es la estructura mínima para el desarrollo de una API basado en un modelo, que a su vez está conectado con la base de datos a través del ORM.

Normalmente la lógica no suele ser tan simple y tanto el serializador, como el view set admiten reescritura de sus funciones para añadir nueva funcionalidad. En cuanto a la entrada de datos a través de POST y PUT admiten validadores para poder devolver errores en el caso de datos incorrectos.

4.9 Autenticación y la autorización

En cuanto la autenticación, se ha usado el sistema propuesto por django. Se ha visto que hay servicios externos que se pueden integrar fácilmente y que resuelve cualquier circuito de autenticación y autorización que nos planteemos. Estamos hablando de auth0 (<https://auth0.com/>).

4.10 Generación de PDF

En cuanto a la generación de pdf hemos probado un servicio basado en plantillas Word, llamado docxmerge (<https://docxmerge.com/es/>). Este servicio permite combinar un fichero json con una plantilla Word y producir un pdf.

4.11 Almacenamiento de ficheros

Cada vez más, los sistemas de información usan servicios para el almacenamiento de ficheros. En las pruebas realizadas se ha usado el software minio (<https://min.io/>)

4.12 El envío de Emails

La generación de emails en desarrollo siempre es un problema ya que si usamos un sistema real tenemos que tener diferentes cuentas.

Se ha usado el software mailhog (<https://github.com/mailhog/MailHog>), para poder simular envíos y validar lo cómo se reciben.

4.13 La gestión de los errores

Se ha usado la integración de los errores con un servicio llamado sentry (<https://sentry.io/welcome/>), que permite recibir los errores y comunicarlos por medios como Slack, mail, etc

4.14 El caché

Se ha usado el software REDIS para probar el almacenamiento de cache dentro de la aplicación.

5 Proceso asíncrono de mensajes

5.1 Descripción del entorno

Celery (7) es un módulo desarrollado en Python que permite la gestión de tareas enviadas a una cola de mensajes, gestionada por RabbitMq o Redis. Celery se integra en Django de manera natural, permitiendo lanzar mensajes a la cola de mensajes. Celery a través de workers realiza estas tareas de manera asíncrona.

COMPONENTES DEL CELERY

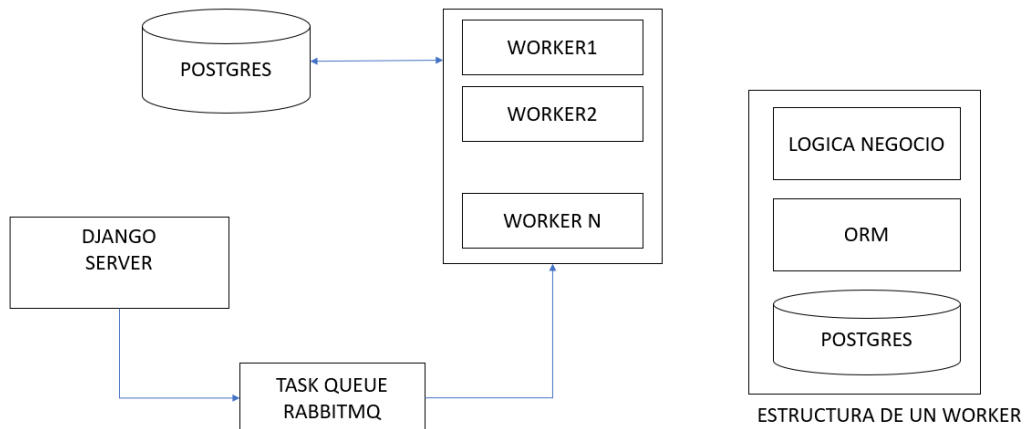


Ilustración 11 Celery

Vemos que desde Django enviamos un mensaje a la cola gestionada por rabbitmq. Este mensaje es procesado por un worker. El worker usará la misma lógica de negocio definido en la web. Esta lógica usará el modelo de datos a través del ORM. Así pues, se reutiliza lógica de la web para uso de los worker.

Usaremos la cola de mensajes para realizar tareas que sean costosas y que hacerlas de manera síncrona aumentaría los tiempos de respuesta de la web. Por ejemplo, transformar una imagen subida a una web, analizar un pdf, enviar un mail, son tareas que llevan tiempo y son apropiadas para que se hagan a través de un sistema de colas.

Celery puede usar varios brokers de mensajes. Es más usado es RabbitMQ (8), pero se integra con otros como REDIS (9).

5.2 Celery con python

El funcionamiento del celery está expresado en la figura siguiente

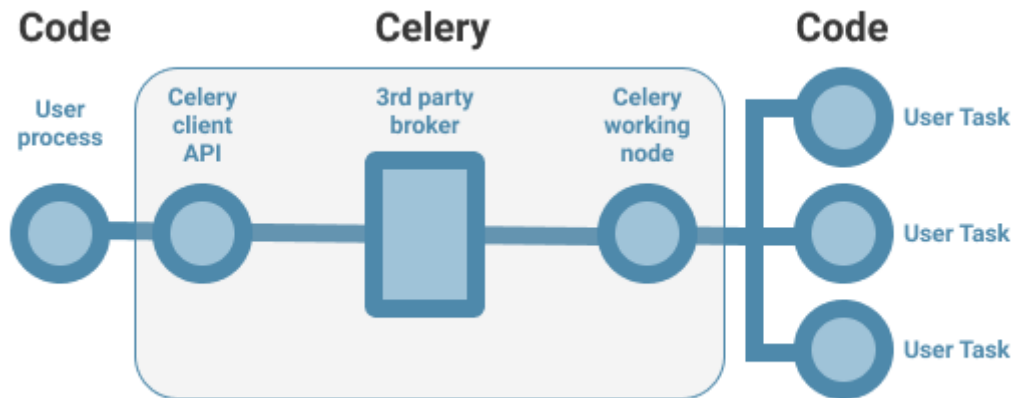


Ilustración 12 Funcionamiento de celery

En la figura vemos que el proceso del usuario, a través de un api, envía un mensaje a el broker. El borker puede ser RabbitMQ. RabbitMQ mantiene los mensajes en colas hasta que un Celery Worker lo procese. El Celery Worker va procesando mensaje y ejecutando el código asociado al mensaje, representado en la figura por User Task.

Instalamos celery con **pip install celery**. Si estamos en Windows debemos de instalar eventlet con **pip install eventlet**. Eventlet (10) es una librería python para manejo de llamadas concurrentes.

```
from celery import Celery
# definición de la aplicación celery
app = Celery('tasks',
             broker='amqp://',
             backend='rpc://',
             )
# definicion de una tarea que se realizará
# cuando haya un mensaje en la cola
@app.task
def add(x, y):
    return x + y
# lanzamiento de la aplicación
if __name__ == '__main__':
    app.start()
```

Lanzamos el proceso celery.

```
celery -A tasks worker -P eventlet
```

El proceso de cliente siguiente mete en la cola mensajes que son procesados y se obtiene la respuesta

```
from tasks import add
from celery import group
```



```

# se lanzar 100 tareas
for i in range(1, 100):
    result=add.apply_async((i, i), )
    print(i, result.get())
# se lanzan 4 tareas dentro de un grupo
numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]

res = group(add.s(i, j) for i, j in numbers).apply_async(
    queue='priority.high',
    serializer="json"
)
# aquí tenemos el resultado de las tareas
print(res.get())

```

En el código anterior el celery cliente api pone 100 mensajes en la cola a través de **apply_async**. Con `result.get` podemos obtener el resultado.

El uso de celery está indicado para tareas costosas en tiempo, como redimensionado de imágenes, envío de mails, etc.

5.3 Celery con django.

Celery es un software que puede funcionar con o sin django. En la arquitectura que estamos definiendo lo usaremos con django. Gracias al aislamiento de la lógica de negocio se puede compartir lógica entre Django y Celery.

Hay que destacar que en ejecución se tendrá una instancia de Django y una o varias instancias de celery. El código de la lógica de negocio y los modelos se comparten entre las instancias.

6 Descripción de aplicaciones Flutter

Flutter es un framework desarrollado por Google para desarrollar aplicaciones móviles multiplataforma. Flutter nace en el 2015 y su versión 1 sale en diciembre de 2018.

La base de Flutter es el lenguaje Dart, diseñado por Google en 2010 y que nace para sustituir al javascript en las aplicaciones web. De hecho, existen frameworks que transpilan código Dart a código javascript, sin tener mucho éxito.

En primer lugar, Flutter está hecho en Dart, el cual destaca por su increíble gestión de los recursos del sistema, el cual es esencial cuando estamos trabajando con dispositivos tan variados como móviles de hace 8 años o móviles de última generación. Las aplicaciones deben de responder adecuadamente en cualquier dispositivo.

Flutter parte de unos principios claros:

- **Productividad de desarrollo:** Tiene que ser fácil y rápido de iterar, cuando se modifique el programa, se debe de ver reflejados los cambios en milisegundos, para que se pueda probar lo antes posible. A esto se le llama **HOT RELOAD** y supone una de las mejores funcionalidades que tienen los entornos de desarrollo flutter.
- **Orientación a objetos:** Los programadores ya conocen la orientación a objetos de otros lenguajes. La curva de aprendizaje del entorno será muy corta.
- **Alto rendimiento:** La experiencia de usuario tiene que ser fluida, rápida y consistente en todos los dispositivos móviles.

Flutter nace para el desarrollo multiplataforma, quiere esto decir que vamos a poder producir con el mismo código aplicaciones para Android y para IOS. Cierto es que estas plataformas tienen devices que se comportan de manera diferente y esto hará que haya algunas particularidades dependientes de la plataforma.

Para empezar a desarrollar con Flutter necesitaremos:

- Un computador con el sistema operativo Windows, Linux o Mac
- Instalar Flutter, los pasos variaran dependiendo del sistema operativo, se puede ver más aquí.
- Si trabajamos con Android, tendremos que instalar el emulador, o bien trabajar con nuestro propio dispositivo
- Tendremos que elegir entre 2 editores, **Visual Studio Code** o **Android Studio**.

En cuanto a la máquina de desarrollo, el ideal es trabajar con Mac ya que en esta máquina podremos compilar para ios y para Android. Si trabajamos en Window o Linux, no podremos preparar la compilación para los.

6.1 Widgets

La interfaz de Flutter está compuesta por componentes llamados Widgets. El concepto es que puedas representar la interfaz y su estado con un árbol de Widgets.

Hay 2 tipos de widgets, los que tienen estado, llamados **StatefulWidget** y los que no tienen llamados **StatelessWidget**.

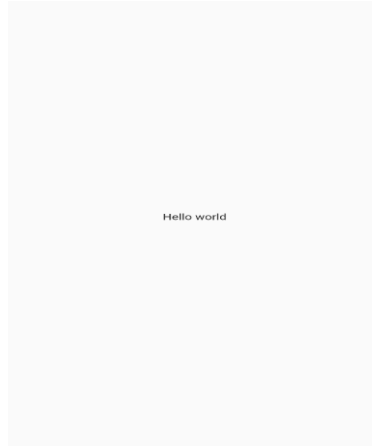
Por ejemplo, si tienes una caja de texto en el que se tiene que guardar el texto introducido, habrá que realizar un Widget de tipo **StatefulWidget**, si, queremos un

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Widget que pinte un círculo rojo, utilizaremos un **StatelessWidget** porque no tenemos que guardar ninguna información.

Vamos a ver paso a paso como seria nuestra primera aplicación en Flutter.

El resultado que queremos obtener es el siguiente:



El resultado de la pantalla anterior, se resuelve con el siguiente que código que repasaremos paso a paso

```
1 import 'package:flutter/material.dart';
2
3 void main() => runApp(MyApp());
4
5 class MyApp extends StatelessWidget {
6   @override
7   Widget build(BuildContext context) {
8     return MaterialApp(
9       title: 'Flutter Demo',
10      debugShowCheckedModeBanner: false,
11      theme: ThemeData(
12        primarySwatch: Colors.blue,
13      ),
14      home: Scaffold(
15        body: SafeArea(
16          child: Center(
17            child: Text("Hello world"),
18          ),
19        ),
20      ),
21    );
22  }
23 }
```

Examinamos la primera línea

```
1 import 'package:flutter/material.dart';
```

Esta es la manera en la que importamos librerías, en este caso, **flutter** es el paquete principal, y **material.dart** es el fichero en lenguaje Dart, que exporta todos los componentes de la especificación **Material**. Si entramos a `material.dart`, podemos ver algunos exportaciones de componentes:

```
1 library material;  
2  
3 export 'src/material/about.dart';  
4 export 'src/material/animated_icons.dart';  
5 export 'src/material/app.dart';  
6 export 'src/material/app_bar.dart';  
7 export 'src/material/app_bar_theme.dart';  
8 export 'src/material/arc.dart';  
9 export 'src/material/back_button.dart';  
10 export 'src/material/banner.dart';  
11 export 'src/material/banner_theme.dart';
```

En el caso de IOS, tenemos la librería de **Cupertino**, la cual tiene un diseño que cumple con el lenguaje de diseño de IOS, se puede ver las especificaciones de este diseño [aquí](#).

Se pueden ver los componentes en la [página de Flutter](#), continuamente van añadiendo nuevos.

La siguiente línea que tenemos es:

```
1 void main() => runApp(MyApp());
```

runApp es una función de Flutter y lo que hace es inicializar la aplicación con el Widget que le hemos pasado como argumento a la función, en este caso, estamos inicializando el widget **MyApp**. Este widget puede ser tanto Stateless como Stateful.

A partir de este momento, solo vamos a trabajar con widgets, estos widgets pueden ser contruidos por nosotros, por Flutter o por librerías de terceros, las cuales se pueden encontrar en [el repositorio de librerías de Dart](#).

El siguiente trozo de código es el siguiente:

```
1 class MyApp extends StatelessWidget {
2   @override
3   Widget build(BuildContext context) {
4     return MaterialApp(
5       title: 'Flutter Demo',
6       debugShowCheckedModeBanner: false,
7       theme: ThemeData(
8         primarySwatch: Colors.blue,
9       ),
10      home: Scaffold(
11        body: SafeArea(
12          child: Center(
13            child: Text("Hello world"),
14          ),
15        ),
16      ),
17    );
18  }
19 }
```

Vamos a ir línea a línea examinándolo.

```
1 class MyApp extends StatelessWidget {
```

En la figura de arriba estamos declarando la clase **MyApp** la cual extiende de una clase llamada **StatelessWidget**, este tipo de Widget como ya comentamos arriba, dados los mismos parámetros produce el mismo árbol de componentes. La clase **StatelessWidget** te obliga a implementar la función **build**, la cual toma como parametro el contexto de tipo **BuildContext**.

```
1 @protected
2 Widget build(BuildContext context);
```

La clase **BuildContext** no es más que una referencia a la localización del widget que estamos renderizando en el árbol de widgets de nuestra aplicación. Se puede utilizar para obtener el alto y ancho del dispositivo, la caja donde se está renderizando el widget actual, interactuar con los widgets padres, etc.

Esta función **build** tiene que retornar un widget.

```
1 @override
2 Widget build(BuildContext context) {
3   return MaterialApp(
4     title: 'Flutter Demo',
5     debugShowCheckedModeBanner: false,
6     theme: ThemeData(
7       primarySwatch: Colors.blue,
8     ),
9     home: Scaffold(
10      body: SafeArea(
11        child: Center(
12          child: Text("Hello world"),
13        ),
14      ),
15    ),
16  );
17 }
18
```

En este caso estamos devolviendo el widget **MaterialApp**, que tiene las siguientes propiedades:

- El título que tendrá nuestra aplicación en el administrador de tareas de Android y IOS
- `DebugShowCheckedModeBanner` indica si se muestra un banner cuando esta depurándose la aplicación
- `Theme`, es la configuración de la parte visual que tendrá todo nuestro árbol de widgets por defecto. Se pueden configurar colores primarios, secundarios, tipos de letra, etcétera. Se puede [ver más aquí](#)
- `home` es el widget principal que se va a renderizar.

En el parámetro `home` tenemos nuestro componente:

```
1 Scaffold(  
2   body: SafeArea(  
3     child: Center(  
4       child: Text("Hello world"),  
5     ),  
6   ),  
7 )  
8
```

Scaffold es un widget que permite diseñar una estructura consistente Material en nuestra aplicación, como, por ejemplo, una barra de aplicación en la parte superior (**AppBar**), un **Drawer** en el lateral, un menú en la parte inferior (llamado **BottomNavigationBar**)

En este caso solo hemos rellenado la parte del **body**, en el que hemos utilizado el widget **SafeArea**, el cual se encarga de renderizar el widget **child** en una parte de la pantalla que sea visible por el usuario. Como **child** de **SafeArea** tenemos un widget llamado **Text** al cual le pasamos el String "Hello world".

6.2 Librerías para cada plataforma

Aunque toda nuestra aplicación este programada en Dart, no significa que no podamos acceder a funcionalidades de cada dispositivo. Para solventar este problema, se utilizan los llamados "**Platform channels**" los cuales nos permiten interactuar con código nativo tanto en la plataforma Android como en IOS.

En un 90% de los casos, podemos encontrar librerías ya desarrolladas por terceros en el [gestor de paquetes de Dart](#), por ejemplo, podemos encontrar librerías hechas por los mismos creadores de Flutter, que van a estar mucho mejor mantenidas y más libres de errores que si las desarrollásemos nosotros.

Pero para acercarnos a cómo sería desarrollar una librería, vamos a hacer una prueba de cómo sería obtener el porcentaje de la batería en un dispositivo Android.

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Para comunicarnos con nuestro código, lo primero que tenemos que decidir es como se va a llamar este canal, normalmente suele ponerse el nombre del dominio al revés y después un separador con una barra (/) seguido del nombre del módulo que estemos desarrollando. En este caso lo vamos a llamar **es.kungfusoftware.tfg/battery**.

La comunicación con la plataforma la vamos a realizar con la librería **services.dart** de Flutter, la cual nos exporta estas funcionalidades que no tenemos con **material.dart**.

```
1 library services;
2
3 export 'src/services/platform_channel.dart';
4 export 'src/services/platform_messages.dart';
5 export 'src/services/platform_views.dart';
```

Para ello en el código Flutter tenemos que importar las siguientes librerías

```
1 import 'package:flutter/material.dart';
2 import 'package:flutter/services.dart';
```

Ahora tenemos que declarar nuestro canal de comunicación en el código Dart

```
1 static const platform = const MethodChannel('es.kungfusoftware.tfg/battery');
```

Para invocar a una función en nuestro canal, hay que tenemos una función llamada **invokeMethod**

```
1 final int result = await platform.invokeMethod('getBatteryLevel');
```

Ahora queda hacer la parte nativa, esta parte la tendríamos que desarrollar para una aplicación real en Android y IOS.

En el fichero **MainActivity.kt** tenemos que realizar las siguientes importaciones para obtener el nivel de batería y configurar el canal.


```

1 package es.kungfusoftware.flutter_tfg
2
3 import androidx.annotation.NonNull;
4 import io.flutter.embedding.android.FlutterActivity
5 import io.flutter.embedding.engine.FlutterEngine
6 import io.flutter.plugins.GeneratedPluginRegistrant
7 import android.content.Context
8 import android.content.ContextWrapper
9 import android.content.Intent
10 import android.content.IntentFilter
11 import android.os.BatteryManager
12 import android.os.Build.VERSION
13 import android.os.Build.VERSION_CODES
14 import io.flutter.plugin.common.MethodChannel
15

```

Y la implementación de MainActivity es la siguiente:

```

1
2 class MainActivity : FlutterActivity() {
3     private val CHANNEL = "es.kungfusoftware.tfg/battery"
4
5     override fun configureFlutterEngine(@NonNull flutterEngine: FlutterEngine) {
6         GeneratedPluginRegistrant.registerWith(flutterEngine);
7         MethodChannel(flutterEngine.dartExecutor.binaryMessenger, CHANNEL).setMethodCallHandler { call, result ->
8             if (call.method == "getBatteryLevel") {
9                 val nivelDeBateria: Int
10                 if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
11                     val batteryManager = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
12                     nivelDeBateria = batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
13                 } else {
14                     val intent = ContextWrapper(applicationContext).registerReceiver(null,
15                         IntentFilter(Intent.ACTION_BATTERY_CHANGED))
16                     nivelDeBateria = intent!!.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) * 100 /
17                         intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1)
18                 }
19                 if (nivelDeBateria != -1) {
20                     result.success(nivelDeBateria)
21                 } else {
22                     result.error("UNAVAILABLE", "Nivel de batería no disponible", null)
23                 }
24             } else {
25                 result.notImplemented()
26             }
27         }
28     }
29 }

```

Vamos a ir línea por línea para entender mejor este código.

En la definición de la clase, vemos que extendemos de la clase **FlutterActivity**, FlutterActivity es la Activity que se ejecuta cuando se inicia una aplicación en Android.

```

1
2 class MainActivity : FlutterActivity() {

```

FlutterActivity ya implementa mucha funcionalidad, nosotros podriamos alterar cierta, haciendo override de ciertas funciones. Pero para registrar los canales de plataforma, lo tenemos que hacer en la funcio configureFlutterEngine, como podemos ver en la documentación del código.

```

1 /**
2  * Hook for subclasses to easily configure a {@code FlutterEngine}, e.g., register
3  * plugins.
4  * <p>
5  * This method is called after {@link #provideFlutterEngine(Context)}.
6  */
7 @Override
8 public void configureFlutterEngine(@NonNull FlutterEngine flutterEngine) {
9     // No-op. Hook for subclasses.
10 }

```

En esta función vamos a hacer la siguiente llamada para registrar otros plugins que hayamos podido instalar de terceros vía el gestor de paquetes.

```

1 GeneratedPluginRegistrant.registerWith(flutterEngine)

```

Ahora vamos a registrar nuestro plugin, al que le vamos a pasar el bus de mensajes *binaryMessenger* y el nombre del canal establecido anteriormente. Y vamos a establecer la funcion a la que se va a llamar cuando se llame desde código Dart.

```

1 private val CHANNEL = "es.kungfusoftware.tfg/battery"
2 MethodChannel(flutterEngine.dartExecutor.binaryMessenger, CHANNEL)
3     .setMethodCallHandler { call, result ->

```

La variable *call*, indica la llamada que se ha hecho, es decir, los parametros y el nombre de la función. La variable *result* indica el resultado que le vamos a retornar al código Dart. A partir de este punto, hay que llamar a "result.success" con el valor de retorno, o bien, "result.error" con el código de error y un mensaje para el usuario. En el caso de que no contemplemos la función, llamamos al método "result.notImplemented" y el que llama tiene que contemplar este tipo de error.

```

1 if (call.method == "getBatteryLevel") {
2   val nivelDeBateria: Int
3   if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
4     val batteryManager = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
5     nivelDeBateria = batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
6   } else {
7     val intent = ContextWrapper(applicationContext).registerReceiver(null,
IntentFilter(Intent.ACTION_BATTERY_CHANGED))
8     nivelDeBateria = intent!!.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) * 100 /
intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1)
9   }
10
11   if (nivelDeBateria != -1) {
12     result.success(nivelDeBateria)
13   } else {
14     result.error("UNAVAILABLE", "Nivel de batería no disponible", null)
15   }
16 } else {
17   result.notImplemented()
18 }

```

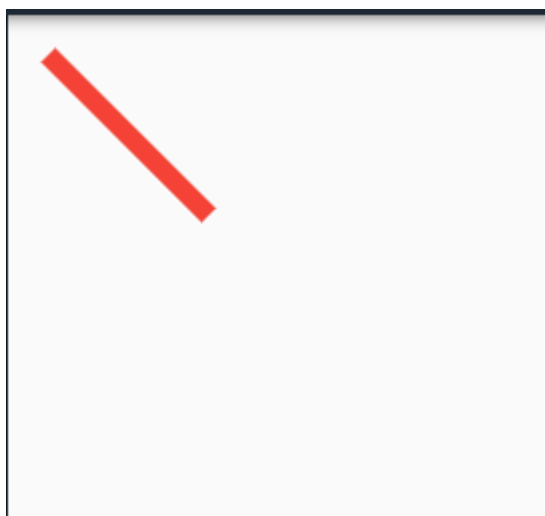
6.3 Renderizado con SKIA

El framework flutter se basa en la librería [Skia](#), la cual se utiliza para renderizar textos, geometrías e imágenes en 2D. Esta librería está escrita en C++, y es multiplataforma, se puede utilizar en Windows, MAC, IOS, Android y Ubuntu.

Podemos hacer pruebas en el playground de [skia.org](#) para poder acercarnos más al renderizado de Skia y para ver hasta qué nivel se utiliza Skia en Flutter.

Tenemos el fiddle donde podemos probar tanto la versión de Skia como la de Flutter.

El resultado que queremos obtener es, una línea roja, con 10 pixeles de ancho que vaya de las coordenadas (20,20) a las coordenadas (100,100). Como la siguiente imagen:



El código C++ es:

```

1 SkPaint p;
2 p.setColor(SK_ColorRED);
3 p.setAntiAlias(true);
4 p.setStyle(SkPaint::kStroke_Style);
5 p.setStrokeWidth(10);
6
7 canvas->drawLine(20, 20, 100, 100, p);

```

Mientras el código flutter equivalente es:

```

1 final Paint paint = Paint(); //SkPaint paint;
2
3 paint.style =
4   PaintingStyle.stroke; //paint.setStyle(SkPaint::kStroke_Style);
5 paint.strokeWidth = 10; // paint.setStrokeWidth(10);
6 paint.color = Color(Colors.red.value); //paint.setColor(SK_ColorRED);
7 paint.isAntiAlias = true; //paint.setAntiAlias(true);
8
9 final Path path = Path(); //SkPath path;
10
11 // equivalente a drawLine(20, 20, 100, 100);
12 path.moveTo(20, 20);
13 path.lineTo(100, 100);
14
15 canvas.drawPath(path, paint); //canvas->drawPath(path, paint);

```

Como podemos ver, tenemos acceso a las funciones de Skia de igual manera en Dart como en C++. Esto nos da una idea de la dependencia que hay entre el framework Flutter y la librería Skia.

Podemos experimentar con estos programas tanto en Flutter como C++ en la web.

Se puede encontrar el código de flutter código aquí

<https://dartpad.dartlang.org/cb708f30cc0b74c1892cb09cd838e852> y el código en C++ aquí <https://fiddle.skia.org/c/aead43a595dc397c0b713bd000a6d176>.

7 Integración y despliegue continuo (CI/CD)

El gran reto del software actual es poder poner cuanto antes en producción nuevas funcionalidades. Para poder hacer esto con seguridad necesitamos herramientas que permitan integrar el software que están desarrollando los programadores y poder generar una versión del código que no tenga errores.

Hablaremos en este capítulo de integración continua y despliegue continuo, que se conocen con el acrónimo (CI/CD). Estas técnicas permiten acortar tiempos y aumentar la fiabilidad del código.

7.1 La integración continua

Entendemos por integración continua a la tarea que realizan los miembros de un equipo de desarrollo orientada a integrar código en un repositorio central, con el fin de poder crear una versión que se pueda testear y descubrir fallo lo antes posible.

Normalmente el entorno de integración continua dispone de un repositorio de código fuente, que suele ser git. Git es el software más extendido para la gestión del código fuente.

Git es un repositorio distribuido. Cada desarrollador tiene en local su propio repositorio en el que va haciendo commit (actualizaciones de código). Cuando su software funciona correctamente y ha terminado la funcionalidad pedida, el programador integra su repositorio con el repositorio central.

Esta integración con el repositorio central puede hacerse con lo que se llama “pull request”. Cuando se hace una pull request, esta no se integra en el repositorio central hasta pasar una validación de otros miembros del equipo. Esta validación suele basarse en ver si el desarrollador ha cumplido las normas del equipo, si el código se comprende por otros participantes del equipo, etc. Si el código no es válido el desarrollador ve los comentarios que han puesto los miembros del equipo, lo corrige y vuelve a subirlo. Cuando la “pull request” es aceptada el código se integra con el repositorio central.

El sistema “pull request” hace que se mejore el código al ser validado por otros miembros del equipo antes de integrarse.

El que el código se haya aceptado no significa que sea correcto y que se pueda generar una versión sin errores.

Una vez que se ha recibido el código de diferentes miembros del equipo, se crea lo que se conoce con el nombre de build. La build es una versión del producto a la que hay que pasar test para ver si todo el código es consistente al menos con los test que tenemos definidos.

Los test dentro del entorno de integración son críticos. Una aplicación en la medida que crece o en la medida que participan más desarrolladores se hace más difícil su consistencia. Solo un buen sistema de test hace que disminuya drásticamente el tiempo de detección de errores y por lo tanto se acelere el proceso de desarrollo.

Ilustramos el proceso de integración continua en la siguiente figura

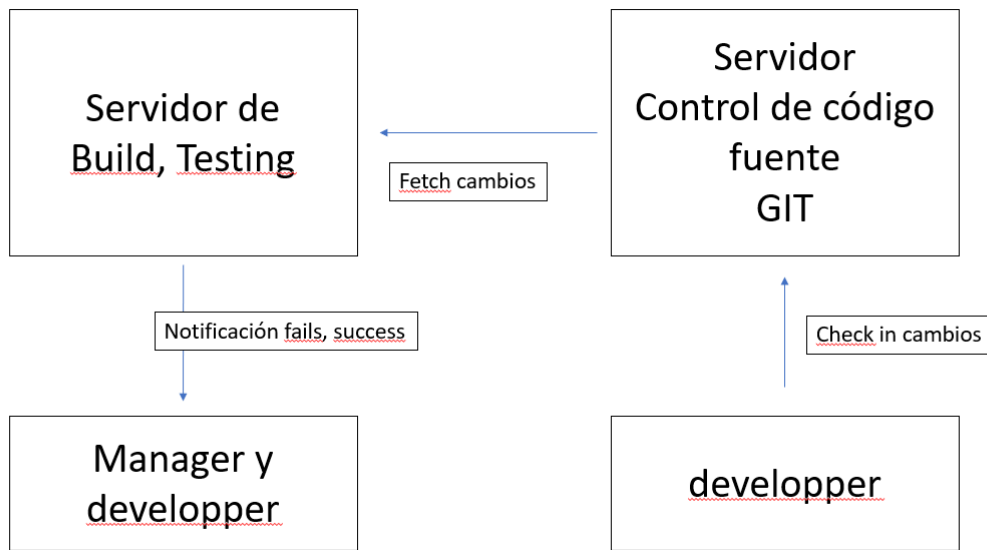


Ilustración 13 Servidor de integración

Lo podemos ver esquemáticamente de la siguiente forma

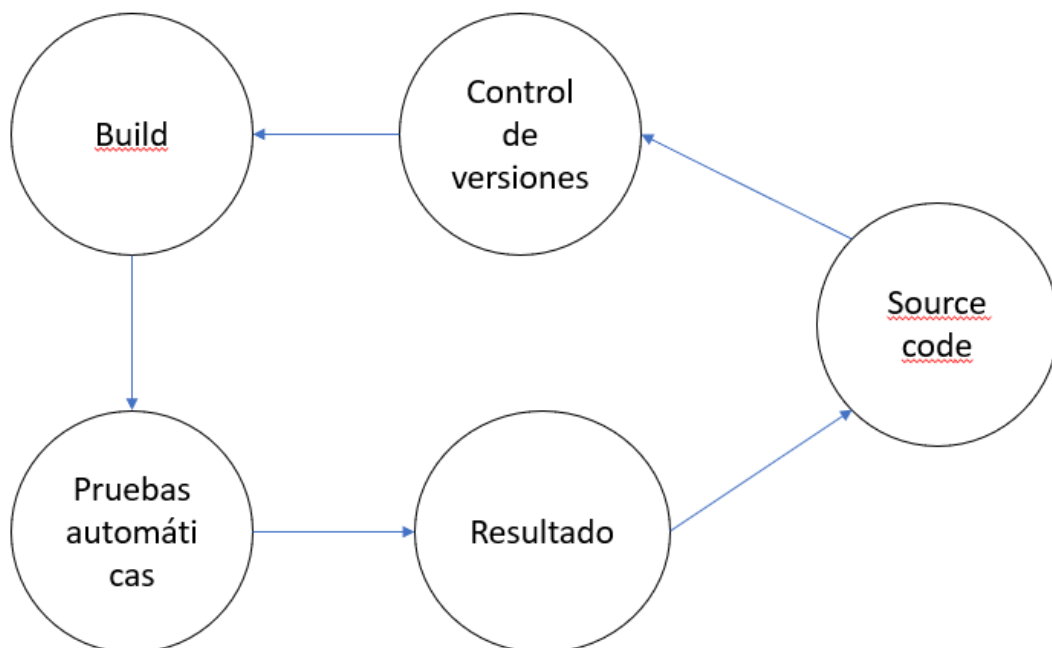


Ilustración 14 Representación esquemática de la integración continua

Se puede apreciar en la figura el funcionamiento iterativo del proceso de integración continua.

Se aprecia que la producción de código fuente, por parte de los programadores alimenta el servidor repositorio de control de versiones. Del control de versiones se obtiene la build. Si la build es construida correctamente, se le pasan las pruebas

automatizadas que producen un resultado. Si el resultado es fallido, el manager y los miembros del equipo han de producir cambios en el código fuente para repetir el ciclo.

Existen muchas soluciones para implementar Integración continua, pero destacaría el producto Gitlab, que es un repositorio de código fuente en el que se pueden definir acciones que permiten realizar la generación de build, pasar pruebas automatizadas y realizar despliegue continuo.

Otra herramienta que tiene un uso en el proceso de integración continua es el SonarQube. Se trata de una herramienta que facilita la calidad de código desde un punto de vista estático y genera indicaciones para mejora. También el software es capaz de detectar código que podría ser vulnerable desde el punto de vista de seguridad. Esta funcionalidad es cada vez más apreciada debido a los ataques más sofisticado por el cibercrimen.

7.2 El despliegue continuo

Por despliegue continuo entendemos el proceso de obtenida una versión y validada según los procedimientos establecidos, esta pasa a producción.

Una vez que tenemos una “build” que ha pasado el proceso de integración, o sea, la creación de la build y las pruebas automáticas, es necesario pasarlo a producción. Si el sistema en grande pueden ser necesarias pruebas adicionales realizadas por un equipo especializado que garantice que la versión esté libre de fallos. Estos equipos llamados equipos de QA han de dar el visto bueno a la versión.

Una vez dado el visto bueno por parte de los responsables se procede a pasar a producción la versión.

Los entornos de explotación más modernos están basados en contenedores. En nuestro caso el procedimiento de pasar una versión de una aplicación web desarrollada en django, significa que debemos de crear un contenedor con el software base necesario para ejecutar nuestra aplicación Django.

Los contenedores son una tecnología reciente que permite crear un entorno de ejecución sin provocar efectos laterales. Anteriormente los entornos de ejecución se metían en máquinas virtuales. Había muchos conflictos entre versiones de código que hacían que fuera complejo el evolucionar las versiones del software.

Con los contenedores, creamos un entorno de ejecución con las versiones de software que necesitemos y esto no va a afectar a otros contenedores, con lo que se consigue un aislamiento.

Si queremos crear un container para ejecutar una aplicación web desarrollada en django podemos usar la siguiente definición del contenedor

```
1 FROM python:3.7
2 ENV PYTHONUNBUFFERED 1
3 RUN apt-get update && apt-get install -y gettext libgettextpo-dev
4 RUN apt-get update && apt-get install -y libgdal-dev
5 RUN mkdir /code
6 WORKDIR /code
7 COPY requirements.txt /code/
8 RUN pip install -r requirements.txt
9 COPY . /code/
```

```
10 EXPOSE 80
11 ENV DJANGO_DEBUG=False
12 ENV DJANGO_SETTINGS_MODULE=gt_web.settings.production
13 CMD python manage.py runserver 0.0.0.0:80
```

El sistema de contenedores es capaz de procesar la definición anterior y crear un contenedor basado en la definición base descrita en la línea 1.

Las líneas 3 y 4 instalan software adicional al container.

La línea 5 crea un directorio llamado code

La línea 7 copia el fichero requirements.txt al directorio code del container.

La línea 8 instala todos los paquetes necesarios para la ejecución de la aplicación

La línea 9 copia el código fuente que tenemos en nuestra máquina al container

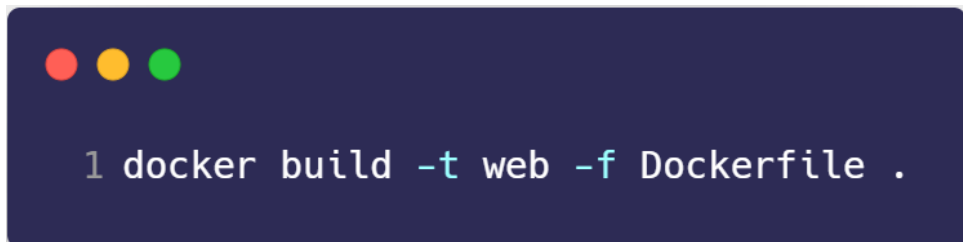
La línea 10 establece que el puerto de trabajo del contenedor es el 80.

La línea 11 establece el Debug a False. El Debug se usa en las máquinas de desarrollo, principalmente.

La línea 12 establece que los parámetros de ejecución de la aplicación son los definidos en la variable `gt_web.settings.production`. Si tuviéramos varios entornos podríamos cambiar esta variable.

La línea 13 establece el Shell de arranque de la aplicación. Se ejecutará cuando arranque el contenedor.

Podemos compilar nuestra imagen con la siguiente instrucción:



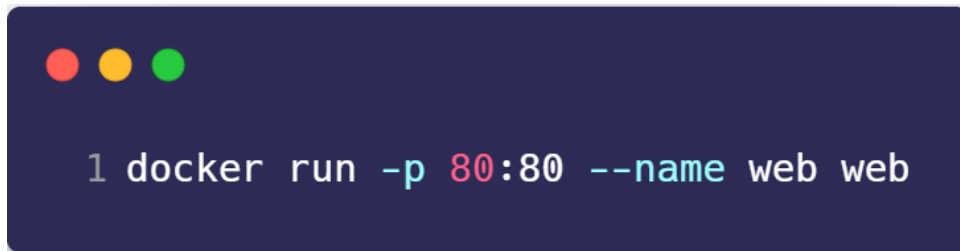
```
1 docker build -t web -f Dockerfile .
```

-t se utiliza como ID de la imagen que vamos a construir

-f indica que Dockerfile usar en esta compilación.

El "." Final, indica el directorio que tendrá acceso las instrucciones en el Dockerfile

Una vez que tengamos esta imagen construida, podemos ejecutarla de la siguiente manera

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal displays the command: `1 docker run -p 80:80 --name web web`. The text is white, with the port mapping `-p 80:80` highlighted in red and the container name `--name web` highlighted in light blue.

Esto ejecutará el comando “python manage.py runserver 0.0.0.0:80”, al levantar el servidor por el puerto 80, podemos relacionar el puerto 80 de nuestro host, con el puerto 80 del contenedor que acabamos de levantar, esto se hace por la opción “-p 80:80”. La opción `--name` significa el nombre del contenedor, el cual tiene que ser único en la máquina que se ejecute.

7.3 Kubernetes como orquestador de contenedores

Cuando tenemos unos contenedores, podemos gestionarlo con comandos de Docker bastante fácil. Pero cuando hablamos de diferentes entornos, escalabilidad horizontal y vertical, múltiples aplicaciones, despliegues sin pérdida de servicio, necesitamos algo más que Docker.

Kubernetes es una plataforma que nos automatiza la ejecución de contenedores a partir de una configuración escrita en formato YAML en la que declaramos como queremos nuestros contenedores. Kubernetes se encarga de que nuestros contenedores se ejecuten correctamente, con todo lo que significa.

Kubernetes está diseñado para funcionar en un entorno distribuido por defecto, por lo que introduce el concepto de nodo, y el concepto de maestro esclavo. Cada clúster de Kubernetes tiene un Master node y el resto son “agentes” que se instalan en otras máquinas y que proporcionan un proxy para que el nodo master lanzar ordenes, como lanzar un contenedor.

La unidad mínima de Kubernetes se llama Pod, el cual es un conjunto de contenedores que necesitan ser desplegados en el mismo nodo.

Un ejemplo de configuración seria la siguiente:

```
1 apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a 1.9.0
2 kind: Deployment
3 metadata:
4   name: web-deployment
5 spec:
6   selector:
7     matchLabels:
8       app: web
9   replicas: 2 # indica al controlador que ejecute 2 pods
10  template:
11    metadata:
12      labels:
13        app: web
14    spec:
15      containers:
16      - name: web
17        image: web:latest
18        ports:
19      - containerPort: 80
```

La anterior configuración nos levantaría la imagen construida antes con “docker build” 2 veces, es decir, si un servidor fuera mal, seguiríamos dando servicio.

Esta configuración se puede hacer grande cuando tenemos muchos contenedores, configuración, diferentes entornos. Para ello podemos utilizar [helm](#), el cual es un gestor de paquetes para aplicaciones de Kubernetes. Helm no es el único, cualquier gestor de plantillas que permita a través de unos valores, generar los ficheros YAML para enviárselos a Kubernetes sería suficiente.

7.4 Despliegue de aplicaciones móviles

Para desplegar vamos a utilizar [Fastlane](#), el cual es una herramienta para automatizar los despliegues a las tiendas de Apple y Google.

Para desplegar vamos a tener que instalar esta herramienta en el runner de Gitlab, que idealmente será un Mac porque podremos desplegar tanto en App store como en la Play Store. Podemos ver en la página de gitlab [instrucciones de como instalar el runner en macOS](#).

7.4.1 Desplegando en IOS

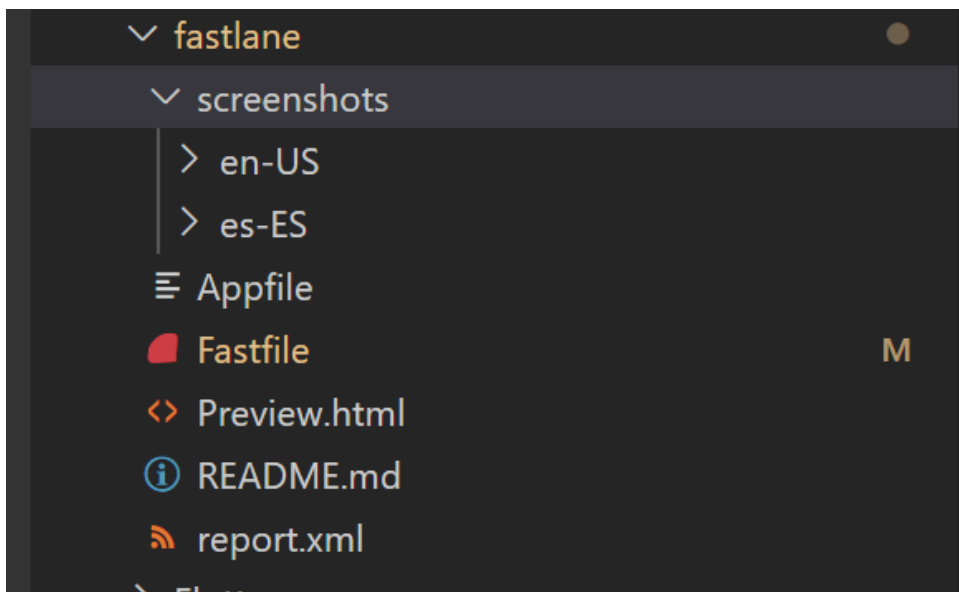
Para instalar fastlane en macOS tenemos que tener Xcode instalado y ejecutar los siguientes comandos.

```
1 # instalar las ultimas utilidades de xcode
2 xcode-select --install
3
4 # instalar fastlane con Homebrew
5 brew cask install fastlane
```

Teniendo en cuenta de que estamos en un proyecto Flutter, nos tenemos que ir al directorio **ios** y ejecutar el siguiente comando para inicializar los ficheros que fastlane necesita para trabajar.

```
1 fastlane init
```

El comando anterior nos generará una carpeta llamada fastlane, que contendrá la siguiente estructura.



El fichero principal es **FastFile** el cual está escrito en **Ruby**, lenguaje principal de **Fastlane**.

Desde el fichero FastFile podemos invocar funciones para desplegar a las diferentes tiendas.

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

Fastlane se compone de un componente llamado *lane*, una *lane* es una función que ejecuta una serie de pasos

```
1 desc "Push a new beta build to TestFlight"
2 lane :beta do
3   cert(username:"davidviejopomata@gmail.com")
4   sigh(username:"davidviejopomata@gmail.com")
5   gym(scheme:"RunnerProd", workspace:"Runner.xcworkspace", export_method:"app-store")
6   upload_to_testflight(skip_waiting_for_build_processing:true)
7 end
```

En este caso estamos realizando una serie de pasos, vamos a verlos mas detalladamente.

```
1 cert(username:"davidviejopomata@gmail.com")
```

En el primer paso, estamos asegurándonos de que hay un certificado válido con el usuario davidviejopomata@gmail.com, si lo hubiese, no crearía ninguno, si no, se encargaría de comunicarse con Apple creando y descargándose un certificado valido.

```
1 sigh(username:"davidviejopomata@gmail.com")
```

En este paso estamos asegurándonos que tenemos un *provisioning profile* válido. Se necesita un *provisioning profile* porque Apple valida que la aplicación que estamos enviando está firmado con nuestro perfil y no con otro, lo que nos protege de posibles ataques en caso de que intentasen publicar una versión modificada de nuestra aplicación.

```
1 gym(scheme:"RunnerProd", workspace:"Runner.xcworkspace", export_method:"app-store")
```

En este paso estamos compilando la aplicación y le estamos diciendo el formato de exportación "app-store", al decirle este parametro, Xcode va a usar el *provisioning profile* de producción, dejándonos un archivo IPA listo para enviarlo a TestFlight o desplegarlo.

```
1 upload_to_testflight(skip_waiting_for_build_processing: true)
```

Por último, enviamos nuestro archivo IPA a testflight, esto lo que hará es subir el fichero IPA y hacerlo público a los probadores que tengamos en nuestra aplicación.

7.4.2 Desplegando en Android

Para desplegar en **Play Store** vamos a tener que ejecutar el siguiente comando en la carpeta **android**

```
1 fastlane init
```

En este caso es mucho más fácil, ya que sólo tenemos que ejecutar el comando de flutter para construir nuestra aplicación.

```
1 flutter build appbundle
2     --build-name ${BUILD_NAME}
3     --build-number ${BUILD_NUMBER}
4     --target lib/main-prod.dart
5     --release
```

AppBundle es un formato nuevo de Android en el que incluimos todo nuestro código y recursos (imágenes, ficheros, videos) y delegamos la generación de la APK y la firma de éste a Google Play. Esto permite que la APK esté optimizada para cada dispositivo, es decir, la APK no será lo mismo para varios dispositivos con diferente versión de Android, esto optimiza el tamaño de la aplicación y por último la experiencia de usuario, al ser más rápida la instalación y las actualizaciones.

ARQUITECTURA DE DESARROLLO WEB CON DJANGO Y APPS CON FLUTTER

El anterior comando nos va a generar un archivo .aab el cual tenemos que subir a google play.

```
1 desc "Release to internal"
2 lane :internal do
3   supply(
4     track: "internal",
5     aab:"../build/app/outputs/bundle/release/app.aab"
6   )
7 end
```

Estamos utilizando [una función de Fastlane llamada *supply*](#) la cual nos sube nuestra aplicación a Google Play. En este caso, estamos diciendo que nos ponga la aplicación en el modo interno. En Google Play podemos definir usuarios a nivel de modo interno y modo Beta.

8 Conclusiones

Este trabajo trata sobre la definición de un entorno de desarrollo de para la construcción de aplicaciones web con Django y de aplicaciones móviles con Flutter.

Se pone de manifiesto la importancia de la calidad del código así como la automatización del proceso de integración de código, los test automáticos y el despliegue continuo.

Por otro lado, se describe el uso de contenedores Docker para gestionar el despliegue de aplicaciones y la integración de estos contenedores en Kubernetes. Kubernetes es una plataforma de orquestación de contenedores que permite escalabilidad, redundancia.

9 Glosario

Android	Sistema operativo para móviles
Android Studio	Entorno de desarrollo oficial para aplicaciones Android
Celery	Celery es un framework para el manejo de tareas basado en colas.
CI/CD	Integración continua / Despliegue continuo.
DAPHNE	En un servidor que permite http(s) y websocket
Django	Framwork de desarrollo basado en Python
Docker	Proyecto que automatiza el despliegue de aplicaciones usando la tecnología de contenedores
Emulator Android / IOS	Sotware que permite la aplicación de un dispositivo virtualizado. Son habituales los emuladores Android y IOS.
ETCD	Sistema de almacenamiento de datos distribuido usado por kubernetes. El nombre viene del directorio etc de los sistema unix añadiendole la letra D de distribuido
Flutter	Entorno de desarrollo multiplataforma para la construcción de aplicaciones móviles
Get	Operación http que solicita una página
GIT	Repositorio distribuido de código fuente.
Http	Protocolo de comunicación usando por navegadores y otros agentes, que permite la comunicación con servidores web
Intellisense	Funcionalidad de los editores que ayuda al programador en la escritura de código
Kubernetes	Entorno orientado a la automatización de despliegues de aplicaciones. Soporte la ejecución de contenedores Docker
Nodo kubernetes	Es un elemento de ejecución de PODS en kubernetes.
Notificaciones Push	Tecnica para enviar mensajes a dispositivos móviles. El dispositivo recibirá el mensaje aunque la aplicación no este levantada
ORM	En el sistema de modelado de base de datos en codigo.
Pod	Un pod ejecuta un grupo de contenedores sobre un nodo de kubernetes.
Post	Operación http que envia datos, provenientes normalmente de una formulario web
Python	Lenguaje de programación creado a principios de los 90.
Rest	Reglas de intercambio de mensajes HTTP que es la base de las API
Template Django	Definición del html con indicaciones especiales, que es invocado por la vistas de django para producir el html que se enviará al navegador
UVICORN	Servidor asincrono en python que permite websocket

Xcode	Entorno de desarrollo de Apple
--------------	--------------------------------

10 Referencias

1. django. [En línea] [Citado el: 10 de 11 de 2019.] <https://www.djangoproject.com/>.
2. <https://flutter.dev/>. [En línea]
3. <https://kubernetes.io/>. [En línea]
4. Django Views. [En línea] [Citado el: 10 de 12 de 2019.] <https://docs.djangoproject.com/en/3.0/topics/class-based-views/>.
5. functions views. [En línea] [Citado el: 11 de 10 de 2019.] <https://docs.djangoproject.com/en/3.0/topics/http/views/>.
6. rest. [En línea] [Citado el: 13 de 12 de 2019.] <https://www.django-rest-framework.org/>.
7. celery. [En línea] [Citado el: 10 de 11 de 2019.] <https://docs.celeryproject.org/en/latest/>.
8. rabbitmq. [En línea] [Citado el: 11 de 10 de 2019.] <https://www.rabbitmq.com/>.
9. redis. [En línea] [Citado el: 11 de 10 de 2019.] <https://redis.io/>.
10. eventlet. [En línea] [Citado el: 11 de 10 de 2019.] <http://eventlet.net/>.