**DOCUMENTO DE ENTREGA**

Este documento constituye una guía de las tecnologías y proyectos realizados hasta la fecha 3 de Enero de 2019.

Junior Paz Formoso

53980855

**ESTRUCTURA DEL MATERIAL ENTREGADO**

**Código fuente**: El código fuente de cada uno de los proyectos desarrollados.

**Configuraciones Correo**: Las configuraciones de correo usadas en este momento en los proyectos.

**Documentos**: Contiene toda la documentación y diseños enviados por cada proyecto.

**Instaladores**: Todos los instaladores principales que se han usado para el desarrollo de los proyectos.

**VPS Documentación:** Documentación de los pasos necesarios para desplegar sitios Django o Laravel en el VPS. La documentación incluye además de documentos, videos y ejemplos del uso de git con bitbucket.

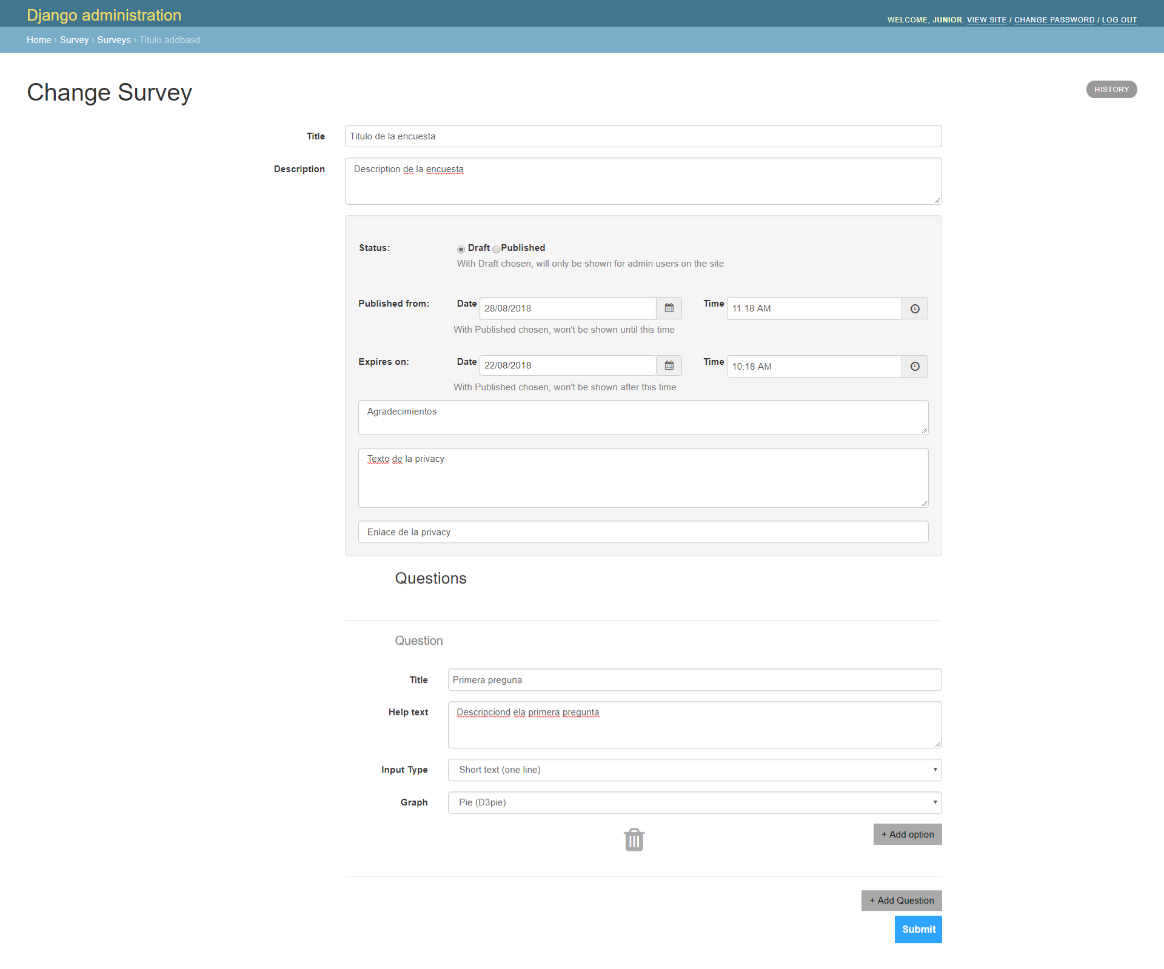
**Documento Entrega.docx**: El documento que estás leyendo.

**TECNOLOGIAS Y VERSIONES USADAS**

1. **Django**: Framework web escrito en Python. Las versiones usadas se describen en los requirements.
2. **Laravel**: Framework web escrito en PHP. Se uso la versión 5.7.
3. **Mezzanine:** CMS Django que agiliza la escritura de aplicaciones Django.
4. **Cartridge:** Sistema Ecommerce que se integra con Mezzanine.

**DESCRIPCION DE LOS PROYECTOS**

1. **Add-b**: Proyecto desarrollado en **Django** que permite generar encuestas. Add-b ofrece un tiempo de diseño de las encuestas, donde se puede crear/diseñar una encuesta que será respondida por un grupo de usuarios vía web. Documento de requisitos Que\_es\_Add-B-ES-001-2018.01.29.pdf



1. **ICONA Milano**: Proyecto desarrollado en **Django** con **Mezzanine**. Este proyecto es una tienda virtual. Las funcionalidades actuales permiten la navegación sobre el catálogo de productos de la tienda. Aun no se encuentra implementada la integración con la pasarela de pago. Documento de requisitos Proyecto-ICONA-Milano-ES-tec.pdf.

Documentos de diseño Archivio-2018.10.26-dev.zip

1. **Beutydates**: Proyecto desarrollado en **Django** con **Mezzanine**. Beautydates hasta el momento es una plataforma sobre la cual se han realizados dos experimentos. El primer experimento conecto Beautydates con Add-b para realizar encuestas. El segundo experimento uso el Ecommerce de Django **Cartridge** para la venta de productos.

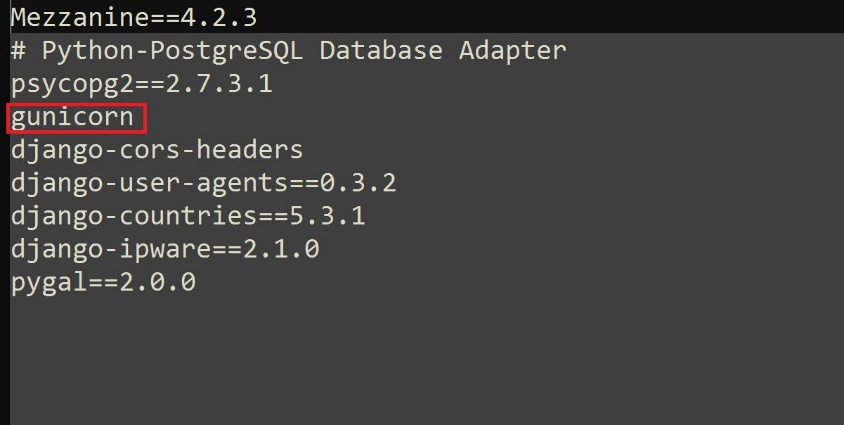
Documento de requisitos segundo experimento 4to-test-v-dev-2018.10.08.pdf.

Documentos de diseños del proyecto actual. Paginas\_web\_Beauty\_Dates.2018.11.30 y beauty-dates-marcos-fotos-polaroid.

1. **Collpack renombrado luego a Felicin**: Proyecto desarrollado en Django, es un pequeño experimento donde se conecto el sistema con Add-b para obtener respuestas por parte de los usuarios a una encuesta diseñada en Add-b. Documento de requisitos segundo experimento Collpack Proyect.txt
2. **Ciberfighters renombrado luego a speekka**: Proyecto que actualmente solo tiene una landing page sin funcionalidad alguna. Este proyecto fue desarrollado en **Laravel**. Documento de requisitos. No existe un documento de requisitos solo los materiales de diseño ya que este proyecto solo es una landing page hasta el momento. Documentos de diseño en business-insight\_home\_boceto-home.2018.11.26.psd y Proyecto ciberfighters/Images.
3. **TheDraft**: Sitio que trata acerca de deporte, específicamente futbol. Desarrollado en Laravel, las funcionalidades actuales involucran una landing page, con un blog y una sección de noticias, así como la integración con **Stripe** para la venta de planes de deporte. Documento de requisitos thedraft.requirements.2018.09.20.pdf. Documentos de diseño ENTREGA TheDraft/TheDraft PC ENTREGA TheDraft/TheDraft CELL y ENTREGA TheDraft/Fonts.

**REQUERIMIENTO PARA LA INSTALACION**

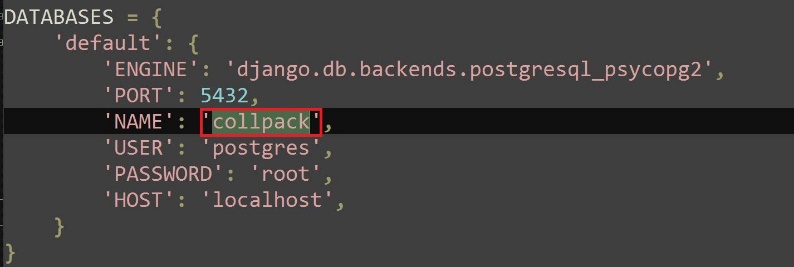
**Requerimientos para Django**. Todos los sistemas desarrollados tienen todos los requirements necesarios para su correcto funcionamiento de forma tanto local como online. En la siguiente foto un ejemplo de los requirements del proyecto collpack.



**gunicorn** es un requirements que en modo alguno es necesario para hacer funcionar el sitio de forma local**,** pero ha sido incluido en los requirements porque si es necesario para publicar el sitio de forma online en el VPS. Lo mismo sucede con todos los requirements algunos paquetes no son necesarios localmente, pero si lo serán para desplegar el sitio.

En resumen, para instalar cualquier proyecto y dejarlo funcionando son necesarios los pasos siguientes: En el video InstalarProyectosEjemploConAddB.avi hay un ejemplo completo de instalación para add-b. Para otro proyecto los pasos son similares.

1. Crear un ambiente virtual
2. Activar el ambiente virtual y ejecutar sobre la raíz del proyecto **pip install -r requirements.txt.**
3. Crear una base de datos de acuerdo a la configuración de base de datos descripta en los ficheros de configuración de Django para el proyecto en cuestión. Por ejemplo, para la siguiente foto seria necesario crear una base de datos de nombre **collpack**

****

Las configuraciones de base de datos estarán siempre en **local\_settings.py**

Importante, crear los ambientes virtuales usando Python 3.6 es la versión usada hasta la fecha en todos los proyectos y la instalada en el VPS. Recomiendo usar cualquier versión en la rama Python 3.X

**Requerimientos para Laravel**. Todos los sistemas desarrollados tienen los mismos requerimientos PHP 7.X y Postgresql.

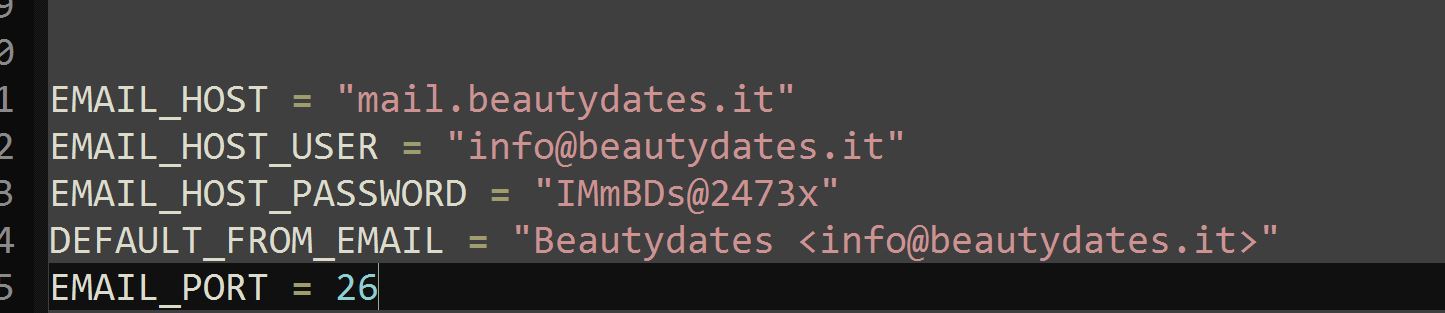
La configuración de base de datos está en el file **.env**

Importante el file **.env** no debería estar versionado, pero lo está y tiene las configuraciones online. **Ojo no cambiar las configuraciones de este file y subirlo al repo.**

Resumen de tecnologías y versiones: Python 3.X, PHP 7.X, Postgresql, Laravel 5.7. De Django se usaron varias versiones. Add-b utiliza la versión 2.0.3 pero Beautydates usa la versión 1.11 esto se debe a que en los proyectos que usan Mezzanine, Django es instalado como parte del proceso de instalación de Mezzanine y la última versión de Mezzanine usa la versión 1.11 de Django.

**CONFIGURACIONES DE ENVIO DE CORREO DESDE DJANGO**

Las configuraciones de correo deberían estar en el file **local\_settings.py** pero están en **settings.py**. Recomiendo quitar las configuraciones sensibles de **settings.py** y moverlas hacia **local\_settings.py.** Es necesario hacerlo directamente en el VPS, el file **local\_settings.py** no esta versionado ni debe versionarse.



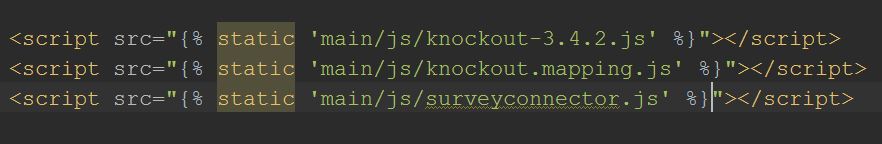
**CONECTAR UN SISTEMA CON ADD-B**

Para consumir las encuestas desde el sistema **Add-b** se desarrollo un conector JS que permite que cualquier sistema web acceda a Add-b para consumir cualquier encuesta diseñada en este sistema.

Para usar el conector son necesarios 3 pasos:

1. Cargar el conector y sus dependencias.
2. Preparar un área de HTML donde se inyectará información para conexión con Add-b.
3. Activar el conector y cargar urls de Add-b.

PASO 1:



Los dos primeros files son dependencias del conector que está desarrollado en knockout.

PASO 2:



Los elementos marcados en rojo son placeholder que se explican a continuación:

**user\_id**: Pk del usuario logeado actualmente

**domain add-b**: Dominio donde este hosteado add-b, actualmente está en <http://add-b.info/>

**survey\_key**: Cuando se diseña una encuesta en add-b se le asigna automáticamente una llave que es única para esa encuesta y que la identifica entre todas, es esta llave la que se pasa como valor del atributo survey.

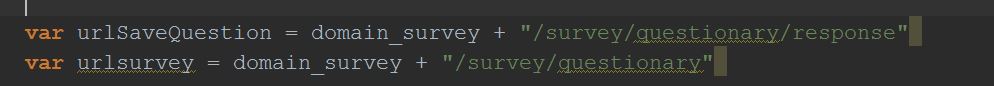
**Url encuesta**: Url donde se está visualizando la encuesta.

PASO 3:

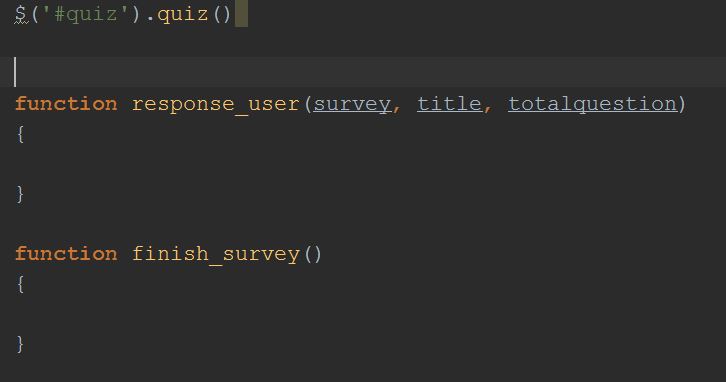
Como parte de la activación deben indicarse los endpoints de Add-b que procesan las respuestas y preguntas. Estos Endpoint devuelven siempre un json como parte de la respuesta. Como uno de los atributos en el paso 2 es el domain entonces estos endpoints pueden construirse dentro del conector, sugiero hacerlo así para evitar declarar siempre estas variables urls de los endpoints que no varían.

**urlSaveQuestion**: Endpoint de Add-b que se encarga de salvar la respuesta que un usuario para una pregunta.

**urlsurvey**: Endpoint de Add-b que contiene la encuesta completa, todas sus preguntas, así como su título, descripción y otros datos que forman la encuesta.



$(“#quiz”) activa el conector, las otras funciones son callback que serán llamados desde el conector en distintos momentos. Ejemplo finish\_survey es llamada cuando la encuesta se responde en su totalidad. Mientras response\_user es llamada cuando un usuario responde una pregunta. Survey será la llave de la encuesta, title el título de la pregunta y totalquestion la cantidad de preguntas respondidas. Dentro del conector no existe verificación para determinar cuando existen o no estas funciones por lo que ahora mismo es necesario definirlas, aunque no tengan implementación.

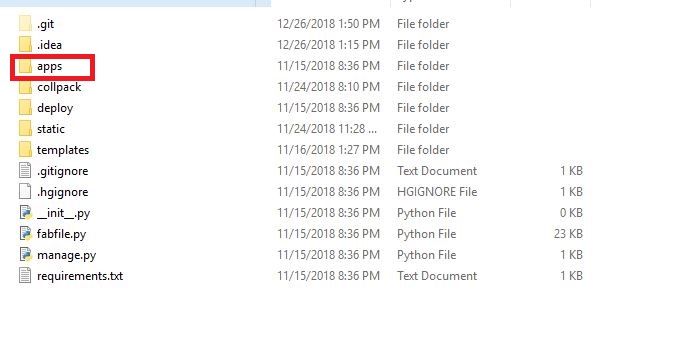


Tener en cuenta que el conector js cubre la mayoría de los casos, pero en algunos casos se necesitan personalizaciones que han llevado a cambiar el conector propio, por ejemplo, en collpack, se necesita de las respuestas que se enviaban a Add-b capturar las dos primeras y llenar el profile del usuario en collpack, de ahí que el conector en collpack ha sido modificado para ello. Recomendación para el futuro. REESCRIBIR EL CONECTOR pues ahora mismo el código no es bueno, cuando se reescriba customizar el conector de forma que existan funciones callback que se llamen en los momentos adecuados de forma que no sea necesario modificar el propio conector según el proyecto. En esta entrega en la carpeta conector esta el conector en su forma más básica posible sin ninguna modificación, de hecho, es el conector usado en el primer experimento y que fue probado en un escenario real.

l

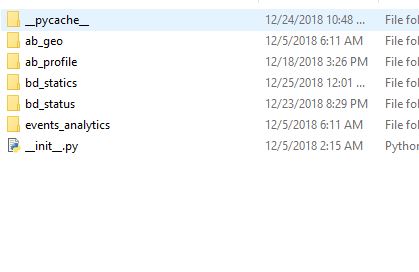
**ORGANIZACIÓN DEL CODIGO EN LAS APLICACIONES DJANGO**

Todos los proyectos responden a la misma organización. Un conjunto de aplicaciones que se encuentran en la carpeta app.

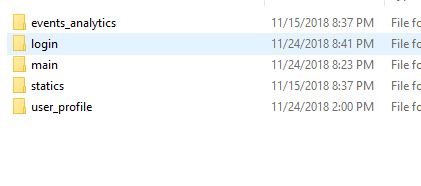
****

**APLICACIONES BEAUTYDATES**

1. **ab\_geo**: Contiene modelos geográficos, para representar País, Ciudad y Código Postal.
2. **ab\_profile**: Contiene funcionalidad relativa el profile del usuario.
3. **bd\_statics**: Modulo de estadísticas que fue usado en el primer experimento.
4. **bd\_status**: Es la aplicación principal donde está la lógica de negocio que se define en cada experimento.
5. **events\_analytics**: Aplicación que permite hacer track de datos como navegador que visita el sitio, desde que sistema operativo, así como registrar todo el flujo de las vistas y paginas en las que se encuentra un usuario. Esta app se usó para las estadísticas.

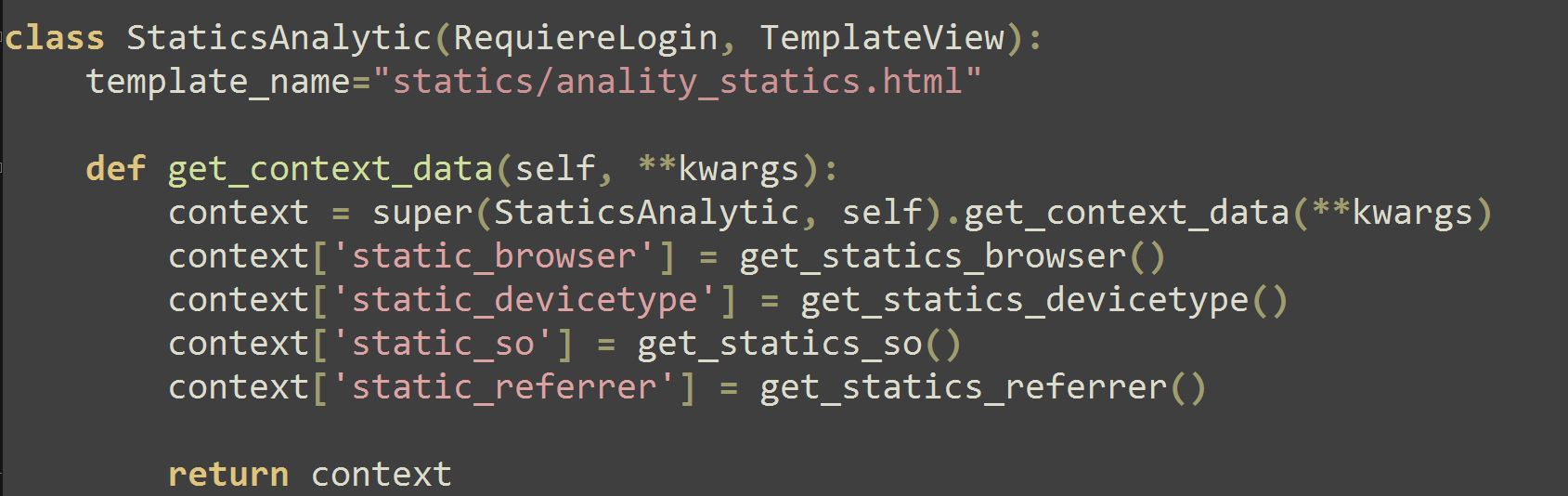
****

**APLICACIONES COLLPACK**



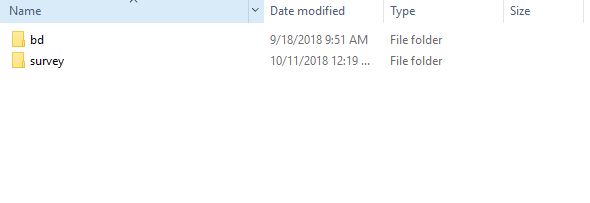
1. **login**: Modulo para la autenticación.
2. **main**: Código de la lógica de negocio del proyecto collpack.
3. **statics**: Estadísticas desarrolladas para el proyecto collpack.
4. **user\_profile:** Modulo del profile del usuario.

Un comentario, si se revisa la app statics se vera el uso de las Class-Based Views de Django.



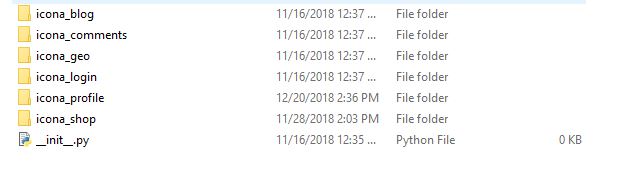
Recomendación siempre que se pueda usar Class-Based View en lugar de las antiguas Functions-Based Views solo usar estas ultimas cuando la complejidad sea muy difícil de lograr con una Class-Based View. En alguno de los proyectos no he usado Class-Based View perdón de antemano, por favor de reescribir esa área; excepto en el caso de Beautydates para ese proyecto en muchos casos esta justificado el uso de los Functions Based View.

**APLICACIONES DE ADD-B**

****

1. **bd**: Haciendo el documento me doy cuenta que es una app donde inicialmente hice algunos test y que ya no es necesaria ni tiene uso alguno, no la he quitado, para entregar el proyecto sincronizado con el repo.
2. **survey:** Es la aplicación que contiene todo el código necesario para permitir la generación de encuestas y permitir recolectar las respuestas de las encuestas que envían los usuarios a Add-b.

**APLICACIONES DE ICONA MILANO**



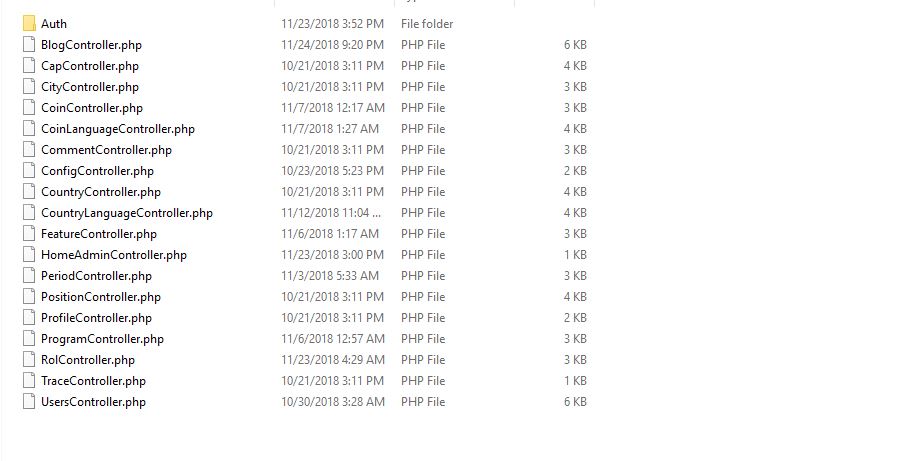
1. **icona\_blog**: Modulo que adiciona la funcionalidad de suscripción de usuarios al blog. Esta app no implementa todo el blog ya que se uso el blog que trae **mezzanine**, esta app solo adiciona comportamiento extra.
2. **icona\_comments:** El sistema de comentarios de mezzanine no trae para postear él phone, el código de esta app es una extensión del propio blog de mezzanine para soportar esta característica.
3. **icona\_shop:** Es la lógica de negocio de la tienda.

Las otras aplicaciones ya fueron explicadas y son reutilizadas en este proyecto.

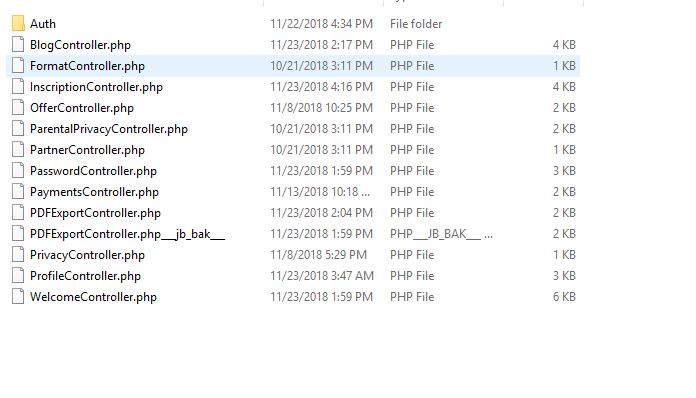
**ORGANIZACIÓN DEL CODIGO EN LAS APLICACIONES LARAVEL.**

En ambos proyectos se respecto el orden por defecto de la distribución Laravel. En el caso del proyecto **TheDraft** en la carpeta **app\Http\Controllers** aparecen las carpetas Admin y Home.

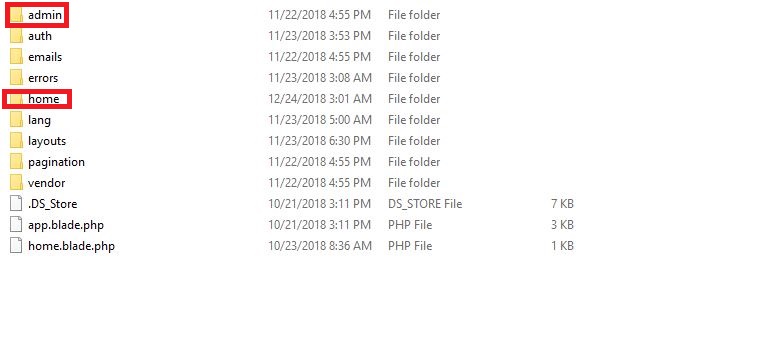
**Admin**: Contienen todos los controladores relacionados a la administración.



**Home**: Contiene todos los controladores relacionados al frontend.



La misma estructura se respecta en los templates/blade existen dos carpetas una de nombre admin para los templates de la administración y otra de nombre home para los templates del home.



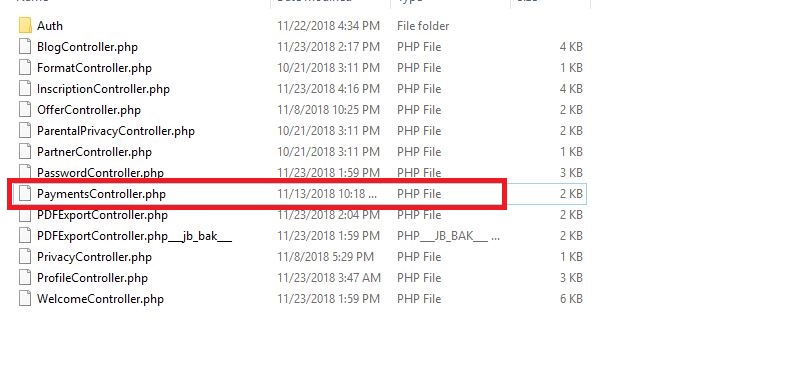
La carpeta layouts contiene los templates base para la administración y para el home.

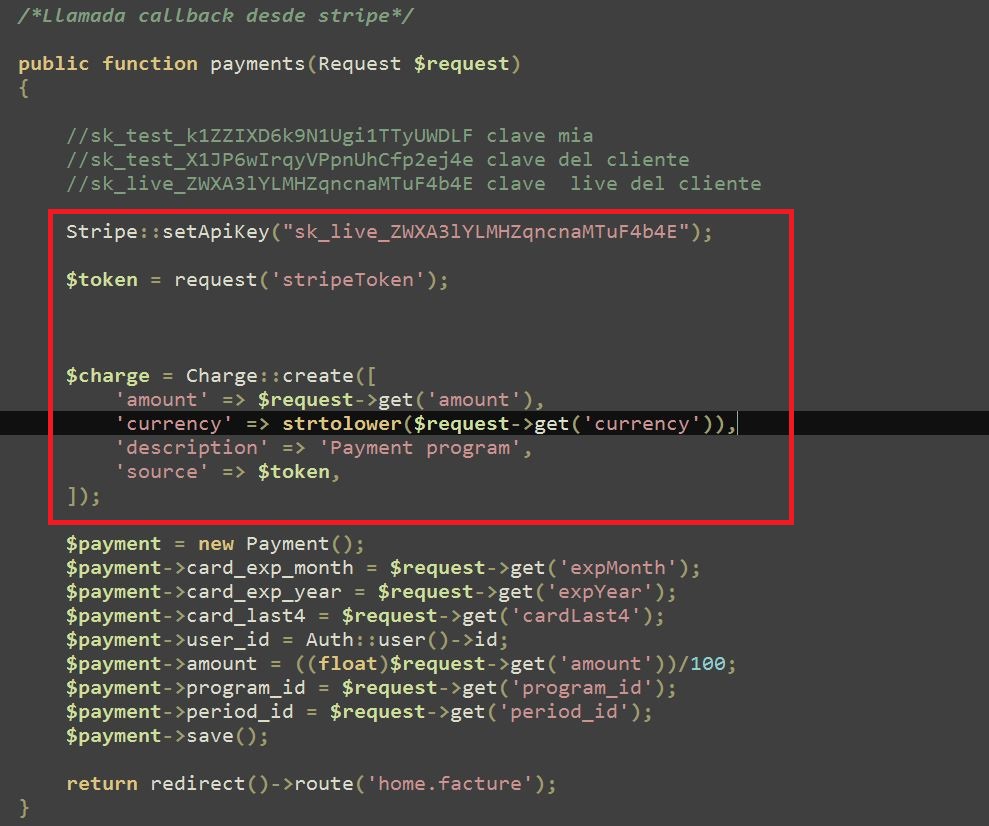
**USANDO STRIPE EN LARAVEL**

Como pasarela de pago se usó Stripe y esta implementado en 2 proyectos. Uno de ellos es el proyecto **TheDraft** y el otro es el proyecto **Beautydates**.

A continuación, se describe el código implementado en Laravel para usar Stripe.

El controlador **app\Http\Controllers\Home\** **PaymentsController.php** y el código necesario para cargar la tarjeta del usuario en el servidor aparece marcado en rojo en las siguientes fotos:





La primera línea de la acción payments del controlador usa la clave live del cliente como llave para cargar la tarjeta del usuario. La segunda línea obtiene el token Stripe que es enviado por Stripe para identificar este pago.

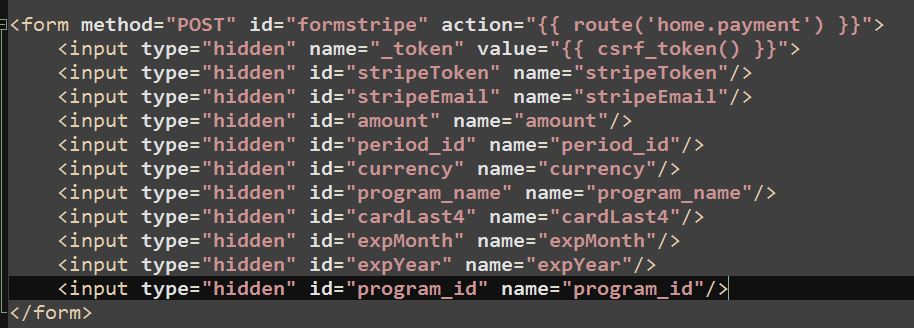
A continuación, se describe el código Stripe en lado cliente y que se puede revisar en el file **resources\views\home\offers\ index.blade.php**. Hay varias formas de usar Stripe, la mas sencilla es usar el formulario por defecto que se puede copiar desde el sitio de Stripe, la segunda forma es customizando el proceso un poco. En este sitio se uso la segunda, pues era necesario capturar los últimos 4 dígitos de la tarjeta que el usuario insertaba, y también llenar en el modal de stripe el correo teniendo en cuenta que el usuario logeado ya tiene correo y es incensario que lo repita de nuevo.

En la siguiente foto se ve como se prepara la configuración de Stripe

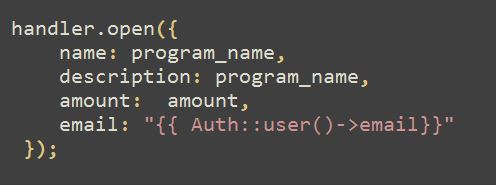


Se indica que se use la clave live publica del cliente. En stripe hay dos llaves una secreta que es la que se vio en el controlador y una publica en el cliente, los prefijos indican de que clave se trata.

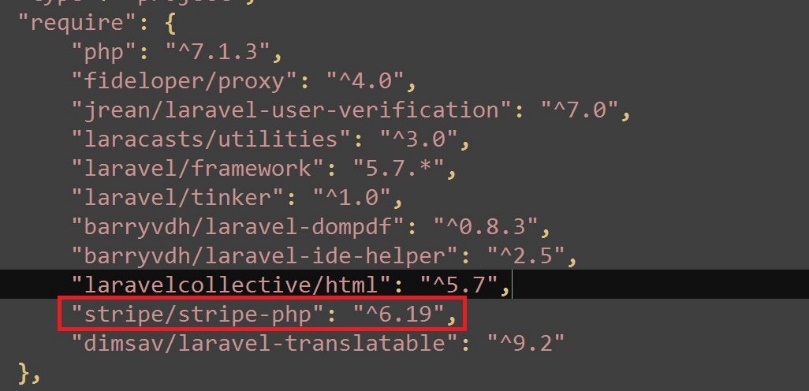
En esta configuración se ha cambiado la imagen que aparece en el modal de Stripe donde el usuario inserta su tarjeta y datos. En la sección token hay una función callback que se llama automáticamente cuando stripe comprueba que la tarjeta es valida y que se puede efectuar la compra para los datos que inserta el usuario; en ese momento stripe llama a esta función automáticamente y pone a disposición en el objeto token los datos de los últimos 4 dígitos de la tarjeta token.card.last4 y otros datos como el email. El único de estos campos que es imprescindible para efectuar la compra es el campo token.id que es el token que se usara en el servidor para cargar la tarjeta del usuario. Los demás son enviados para cumplir requerimientos del proyecto en particular token.card.last4 los otros los he guardado por si se presenta necesario en el futuro alguno de estos datos ya tenerlos. Al final del callback se ejeucta $(“#formstripe”).submit() que envia el siguiente formulario oculto luego de inyectar en el mismo toda la información necesaria.



Todo el proceso anterior describe las configuraciones y el formulario que se enviara, pero no abre el modal en que el usuario inserta sus claves. La manera de abrir este modal es como se describe en la siguiente foto de abajo. Notar que en esta foto se ha llenado el email lo que causa que cuando el modal de stripe abre automáticamente aparece el correo llenado sin necesidad que el usuario tenga que escribirlo.



La librería de Laravel que se uso para comunicarse con Stripe desde el servidor aparece marcada en rojo en la foto siguiente;

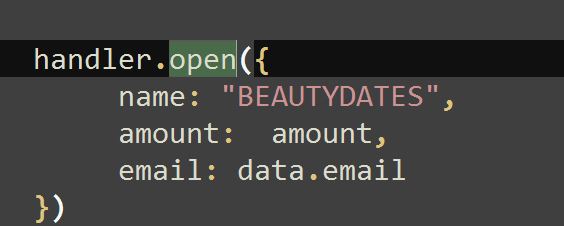


**USANDO STRIPE EN DJANGO**

La lógica y código necesario en el lado cliente no se describe pues es la misma usada en Laravel, de ser necesario revisar el file apps\bd\_status\static\bd\_status\js\cart.js en este file aparece el código, pongo adjunta dos fotos a manera de resumen



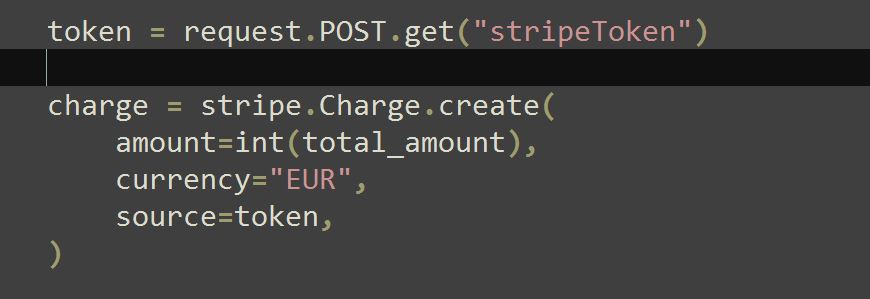
Esta es la sección relacionada a la configuración mucho mas sencilla esta vez pues la única customización es el prellenado del campo email que sucede como se ve en la siguiente foto cuando se manda abrir el modal de stripe.



El siguiente es el código en el lado servidor y que se encuentra en la view beautydates\apps\bd\_status\view\_register\_user

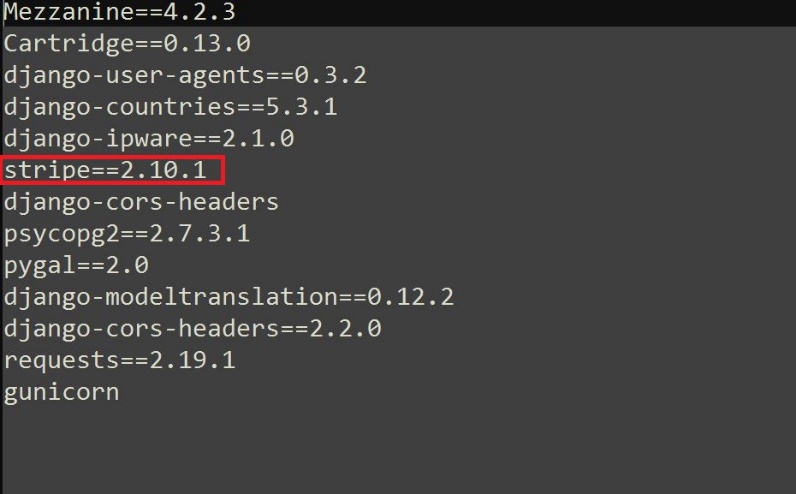


Primero se importa Stripe y se indica que se use la llave secreta que se encuentra en los settings, esta información que es sensible forma parte de la información que sugiero mover hacia el file local\_settings.py de forma manual en el VPS, es solo una sugerencia lo dejo a consideración y según el acceso y condiciones de Internet para estar conectado en el VPS.



Este es el código que carga finalmente la tarjeta del usuario.

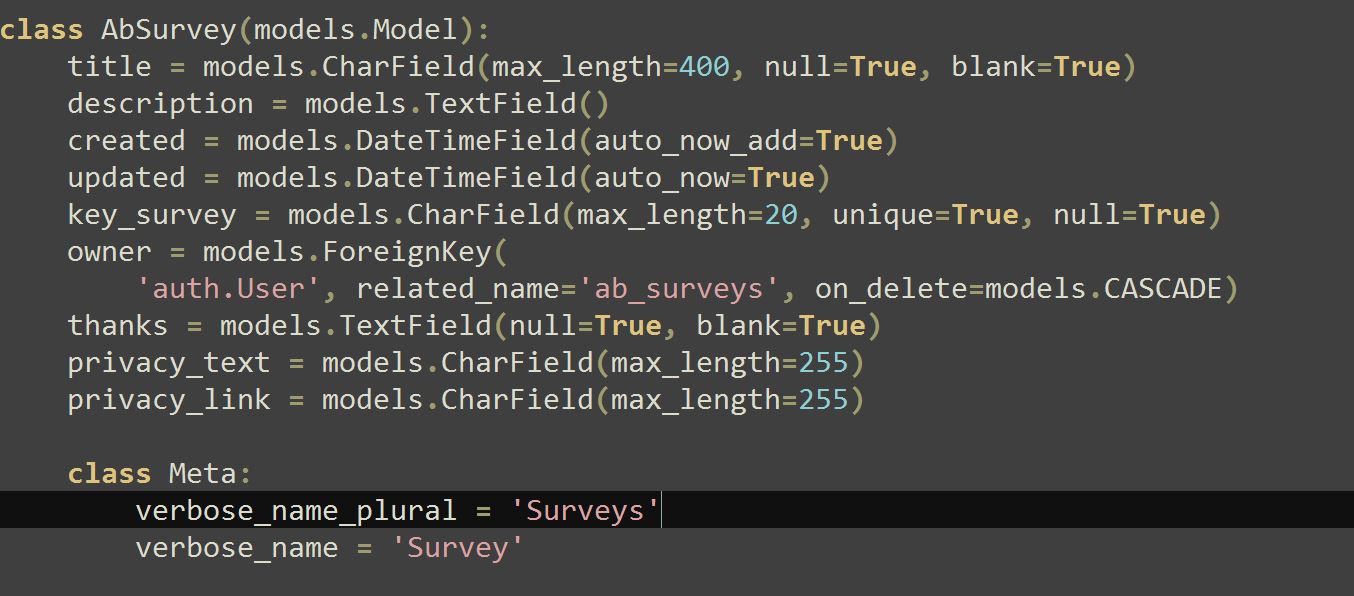
La librería que se uso en Django para conectarse con Stripe aparece marcada en rojo en los requirements que se ven en la siguiente foto



Aprovecho la oportunidad para comentar que la versión de Django ni el propio Django aparecen reflejados en ninguno de los proyectos que usan Mezzanine pues como parte de la instalación de Mezzanine se instala Django que es una dependencia.

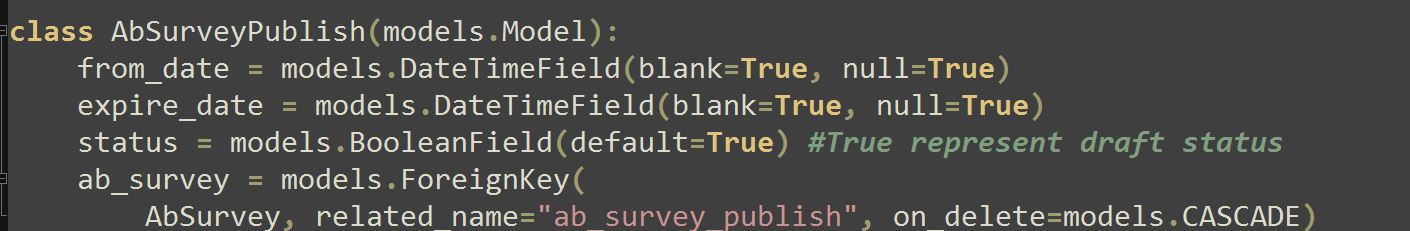
**DESCRIPCION DE LOS MODELOS EN EL PROYECTO ADD-B**

MODELO AbSurvey



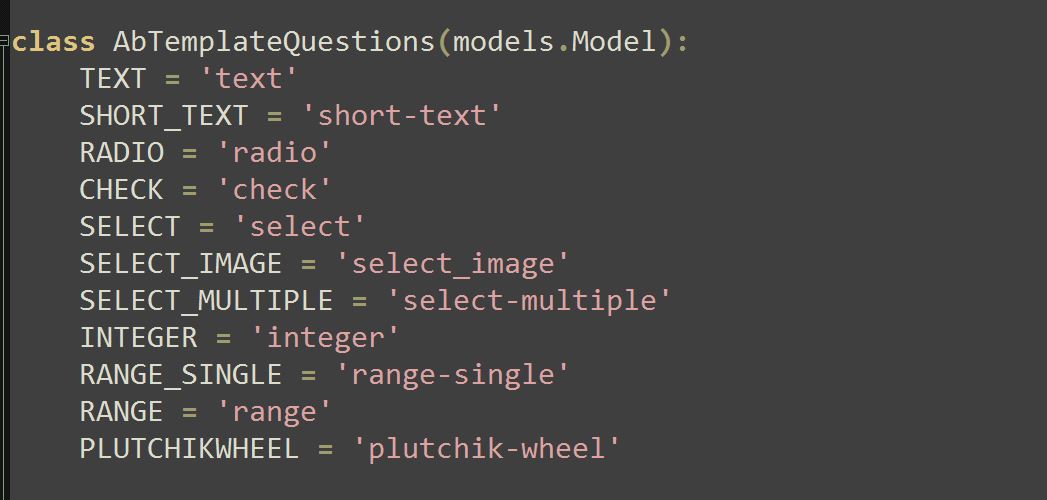
Define una encuesta, definiendo su título, descripción y otra información de importancia. Destacar el campo **key\_survey** es el campo que es único para toda encuesta.

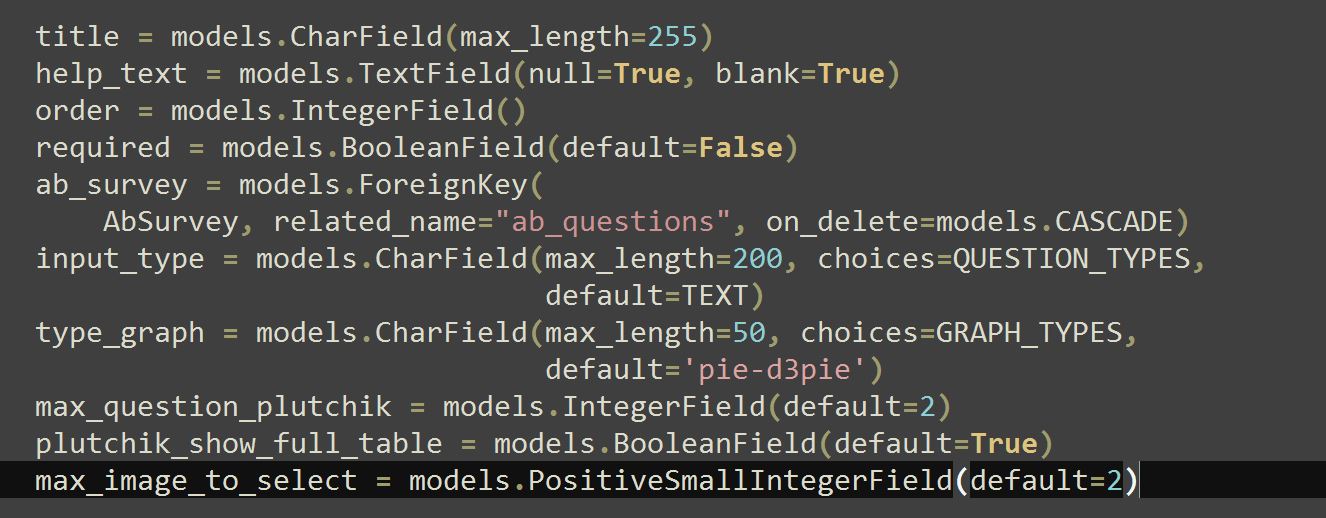
MODELO AbSurveyPublish



Define la información de cuando expira una encuesta, En estos momentos en ninguno de los proyectos se está explotando esta información.

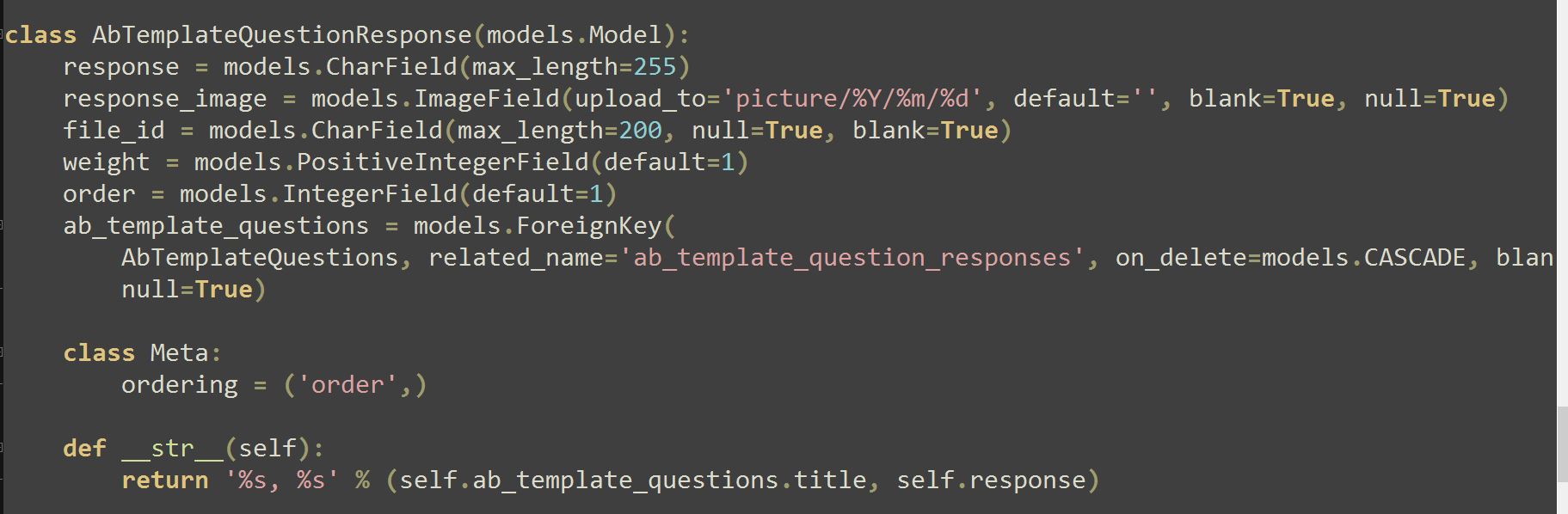
MODELO AbTemplateQuestions





Este modelo representa a una pregunta la cual queda definida por un título, descripción, tipo de pregunta, las preguntas pueden ser textuales, de texto corto o texto largo, preguntar de selección usando select, o preguntar numerics que permiten solo usar números y otros tipos de preguntas que quedan definidas en el documento **boceto-gestor-de-encuestas.pdf**. El campo **input\_type** define el tipo.

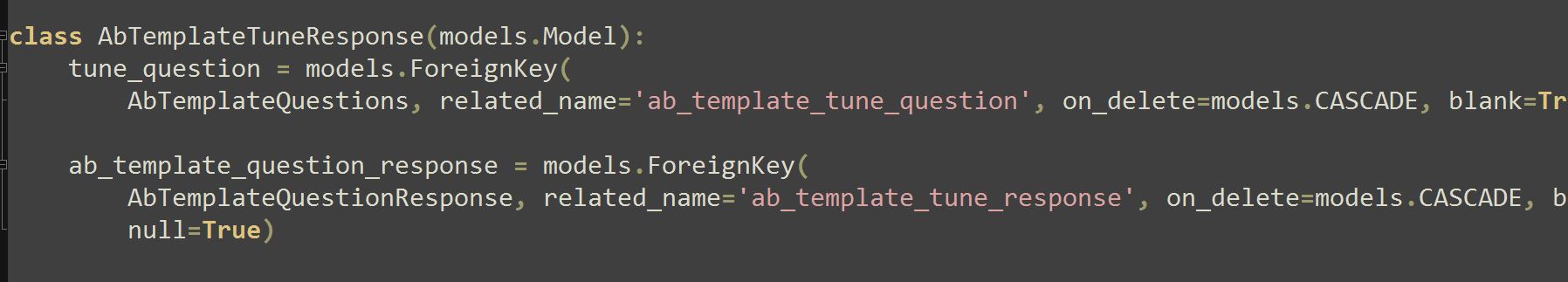
MODELO AbTemplateQuestionResponse



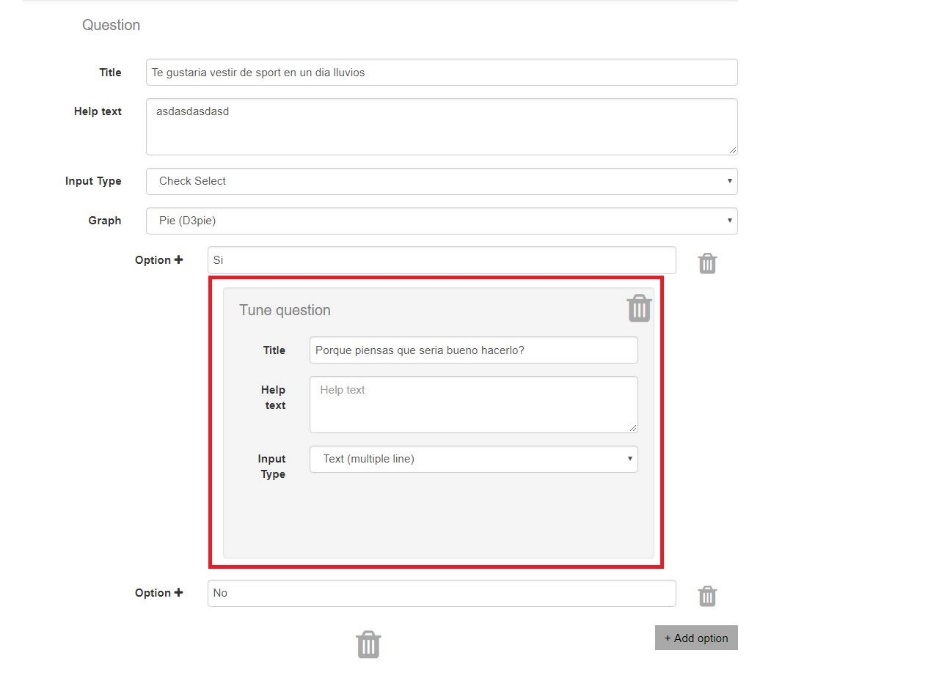
Este modelo representa las posibles respuestas que podrá dar un usuario a una pregunta, esta relacionado con una pregunta a través del campo ab\_template\_questions hay 3 campos response, response\_image y file\_id pues algunas preguntas son de tipo image o file las otras serán de tipo texto caso donde se usara el campo response.

weight actualmente no es explotado y se usara para hacer minería de datos sobre las respuestas de las encuestas

Los modelos anteriores representan la definición de una encuesta de sus preguntas y las respuestas posibles a cada pregunta. Aclarar que algunas preguntas son de tipo libre y la respuesta no se formaliza en el modelo AbTemplateQuestionResponse, este modelo solo será usado cuando se defina que tipos de respuesta se espera del usuario lo que sucede en las preguntas de tipo checkbox por ejemplo donde el usuario debe marcar una opción y las opciones son las respuestas posibles a seleccionar.

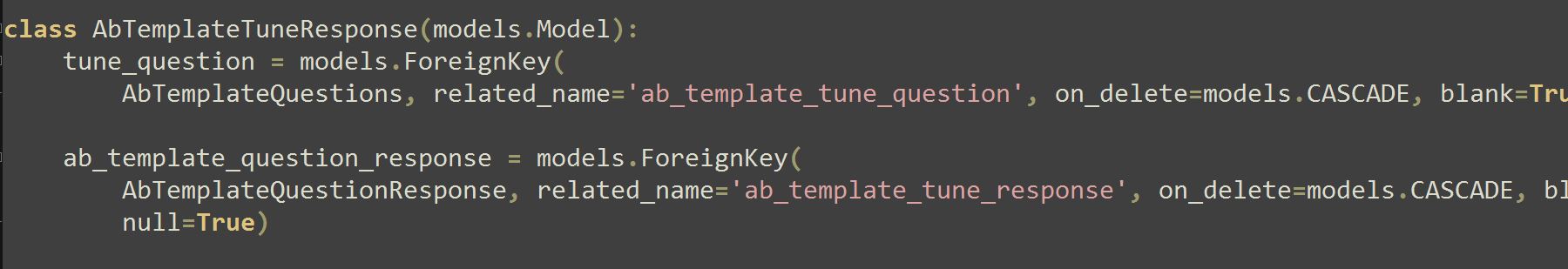


Según la respuesta que un usuario de a una pregunta existe la posibilidad de refinar la pregunta. Por ejemplo, en caso de preguntar algo con posibles respuestas de si o no, es posible refinar la pregunta Se muestra una foto ejemplo del tiempo de diseño de Add-b donde sucede esto. En caso de responder Si se realizara otra pregunta

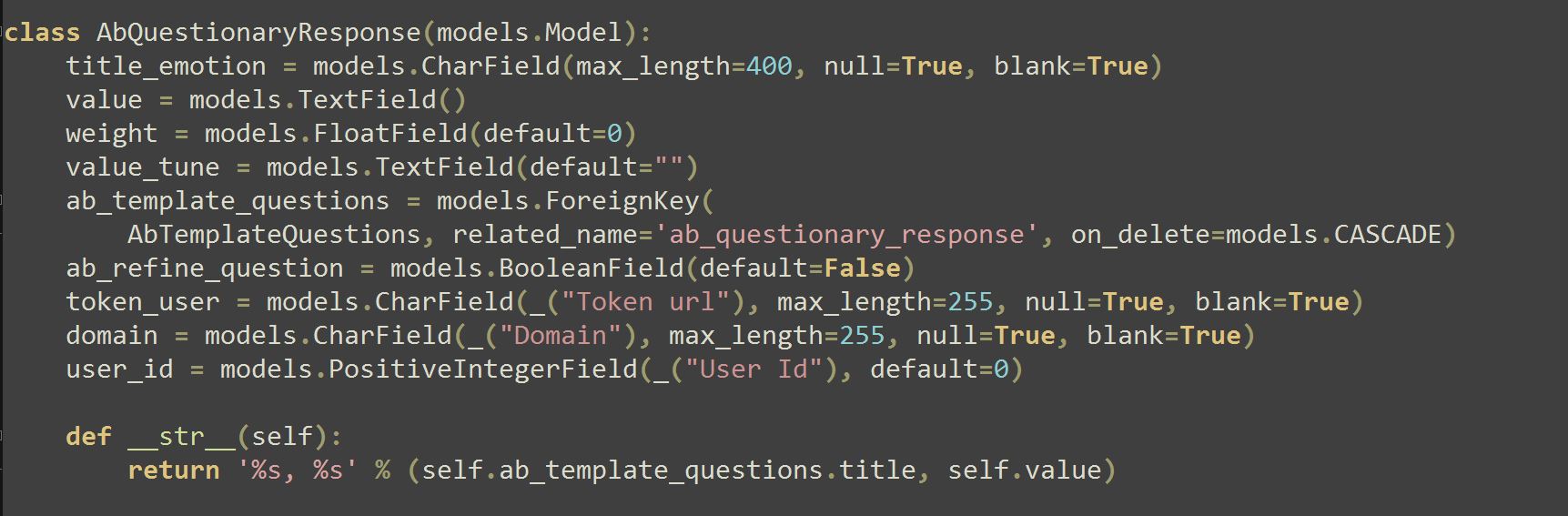


El Modelo siguiente representa este tipo de preguntas que se hacen en base a respuestas.

MODELO AbTemplateTuneResponse



Los siguientes modelos representan las respuestas que se capturan cuando un usuario responde una encuesta.



Este modelo representa la respuesta a una pregunta. user\_id representa el id del usuario que está respondiendo y domain el sitio que en ese momento está consumiendo el servicio ofrecido por add-b.

**PROYECTO BEAUTYDATES EXPERIMENTO 2**

Este proyecto usa como documentos de requerimientos los siguientes:

1. Documentos/ Proyecto BeautyDates/ Segundo experimento/4to-test-v-dev-2018.10.08.pdf
2. Documentos/ Proyecto BeautyDates/ Segundo experimento/ Indicaciones.txt
3. Documentos/ Proyecto BeautyDates/ Segundo experimento/ pacchi-colettivi-test.pdf

Este proyecto tiene integración con Stripe, como parte de la entrega muestro en un video como desactivar Stripe para poder probar las funcionalidades offline. Recomiendo poner una variable en los settings y según ella activar o desactivar Stripe, en el video muestro que áreas de código deben desactivarse. Este proyecto se encuentra en desarrollo en este momento, así que tratare de explicar el flujo de navegación, las distintas paginas involucradas y notificaciones. Empecemos con los siguientes videos.

**Desactivar Stripe y realizar pago.avi**: En este video se muestra como desactivar Stripe y realizar una compra de forma offline.

En este experimento se distinguen dos tipos de usuarios, aquellos que son miembros y aquellos que llegan por primera vez como invitados a la plataforma. Un usuario miembro es aquel que cumple alguna de las siguientes condiciones.

1. En el modelo **Profile** el campo **has\_status\_beauty\_member** tiene valor True. Notar que el modelo Profile está relacionado con un usuario. Este modelo indica el perfil del usuario y tiene otros campos más que no se muestran aquí.

**apps/ab\_profile/models.py**

class Profile(models.Model, AdminThumbMixin):  
 *"""  
 A profile.  
 """* user = models.OneToOneField("auth.User", related\_name="Profile")  
 has\_status\_beauty\_member = models.BooleanField(default=False)

1. El usuario tiene asociado un **UserStatus** a través del campo user del modelo **UserStatus**. Este modelo tiene otros campos que no se muestran aquí.

**apps/bd\_status /models.py**

class UserStatus(models.Model, AdminThumbMixin):  
 user = models.ForeignKey("auth.User", on\_delete=models.CASCADE, related\_name='StatusUser')

El resto de los usuarios que no cumplen ninguna de estas condiciones son usuarios invitados.

Existe un proceso de notificación cuando un usuario invitado hace compras, la notificación es la siguiente:

Se envía un correo indicándole al usuario miembro que lo invito de que una amiga invitada a realizado una compra. En el siguiente video se ve como un usuario invitado hace una compra y como el usuario que le envía el enlace de invitación recibe una notificación.

**MODELO EN BEAUTYDATES**

A continuación, se describen los modelos más importantes en Beautydates.

**MODELO STATE**

class Status(models.Model, AdminThumbMixin):  
 name = models.CharField(\_("Name"), max\_length=100)  
 description = RichTextField(\_("Description"), null=True, blank=True)  
 points = models.IntegerField(default=0)  
 image = models.FileField(upload\_to='status/%Y/%m/%d', null=True, blank=True)  
 created = models.DateTimeField(\_('Created'), auto\_now\_add=True)  
 updated = models.DateTimeField(\_('Updated'), auto\_now=True)

Este modelo define los distintos status que existen en el sistema. Estos status serán asignados a los usuarios. A continuación, se explican los distintos campos del modelo.

**name**: Nombre del status

**description**: Descripción del status.

**points**: Puntos que tiene este estado. Estos puntos definirán los puntos de un usuario que tenga asignado este status.

**image**: Imagen asociada al status. Estas son las actuales tarjeticas que se vienen usando desde el experimento1 y que representan los status Beauty Member, Beauty Girl y Beauty Influence Girl. En las foticos se ven los puntos de cada status.



**created**: Campo que indica la fecha de creación del status.

**updated**: Campo que indica la fecha de la última modificación del status.

**MODELO USERSTATUS**

class UserStatus(models.Model, AdminThumbMixin):  
  
 state = models.CharField(choices=STATUS\_CHOICES, max\_length=100, null=True, blank=True)  
 about\_status = RichTextField(\_("About status"), null=True, blank=True)  
 user = models.ForeignKey("auth.User", on\_delete=models.CASCADE, related\_name='StatusUser')  
 status = models.ForeignKey(Status, on\_delete=models.CASCADE, null=True, blank=True)  
 prize = models.ForeignKey(Prize, null=True, blank=True, verbose\_name=\_("Prize"), related\_name="userStatus")  
 product = models.ForeignKey(ProductVariation, null=True, blank=True, verbose\_name=\_("Product"))  
 collpack = models.ForeignKey(Collpack, null=True, blank=True)  
 product\_won = models.BooleanField(default=False)  
 send\_to\_address\_profile = models.BooleanField(default=False)  
 created = models.DateTimeField(\_('Created'), auto\_now\_add=True)  
 prize\_operator = models.BooleanField(default=True) #Rename to userstatus operator, este campo indica cuando el user\_status lo creo la operadora  
 send\_email = models.BooleanField(default=False)  
  
 address = models.CharField(\_("Address"), max\_length=100, null=True, blank=True)  
 country = models.ForeignKey("ab\_geo.Country", null=True, blank=True, verbose\_name=\_("Country"))  
 city = models.ForeignKey("ab\_geo.City", null=True, blank=True, verbose\_name=\_("City"))  
 cap = models.ForeignKey("ab\_geo.Cap", null=True, blank=True, verbose\_name=\_("Cap"))

Este modelo define el status de un usuario en el sistema. A continuación, se explica cada uno de los campos.

**state**: Campo que representa el estado de envió del producto asociado al UserStatus, puede tomar los valores (**Unprocessed**, **Processed**, **Delivered**). Este campo se sincroniza con el de la Orden. Es decir si se cambia el valor de state la orden/s del usuario asociado quedaran sincronizadas al mismo valor.

**about\_status:** Usado en el primer experimento. Este texto se usaba en el correo de notificación que se enviaba cuando cambiaba el campo state del UserStatus. EN EL SEGUNDO EXPERIMENTO NO SE USA.

**user**: El Usuario al cual pertenece el UserStatus.

**status**: Relaciona al UserStatus con el modelo State explicado anteriormente.

**prize**: Representa los premios/productos que obtenían los usuarios en el primer experimento. EN EL SEGUNDO EXPERIMENTO NO SE USA.

Antes de explicar el campo product algunas explicaciones:

Cuando un usuario gana un regalo cambio su status al siguiente o adquiere el estado **Beauty Member** de no tenerlo en ese momento. Es decir, si su status es **Beauty Member** pasa a **Beauty Girl** y si es Beauty Girl pasa al status **Beauty Influence Girl.** EL cambio de estado es acumulativo, es decir el usuario tiene todos los estados anteriores más el ultimo estado que se crea en el momento de ganar el regalo.

**product**: El campo representa el modelo **ProductVariation** del Ecommerce, es decir los productos que el usuario puede comprar u obtener como regalo. Este campo se llena cuando se crea un UserStatus de acuerdo a la descripción de cambio de status y se llenara con el producto que obtiene como regalo. De existir varios productos regalos en el carrito durante el momento de creación del UserStatus se asigna cualquiera de ellos en el campo producto. Sergio el cambio de estado lo puedes ver en la función change\_status\_when\_gain\_prize

**collpack**: Representa el collpack al que se asocia este UserStatus. El modelo Collpack se explica a continuación y representa el paquete colectivo como una alternativa de compra cuando el envió no es personal.

**product\_won, send\_to\_address\_profile, created, prize\_operator**: Ninguno de estos campos es importante para este experimento.

**address, country, city, cap**: Información de ubicación que se usa para saber a dónde enviar el product. En resumen, es la información geográfica que en este experimento se llena en el modal u que también está asociada al profile.

**MODELO COLLPACK**

class Collpack(models.Model):  
 state = models.CharField(choices=STATUS\_CHOICES, max\_length=100, null=True, blank=True, default="Unprocessed")  
  
 user = models.ForeignKey("auth.User", on\_delete=models.CASCADE, null=True, blank=True)  
 open = models.BooleanField(default=True)  
 message\_mail = models.TextField(null=True, blank=True)

Modelo usado en el método de pago colectivo y que define todos los productos que forman parte de un paquete de productos que será entregado en la casa de una persona, a donde irán todos los usuarios a buscar sus productos. El paquete se enviará a la casa de la persona dueña del paquete colectivo que es la persona que envía el enlace de invitación a sus amigas.

A continuación, se describen los campos:

**state**: Este campo es el mismo que el analizado en UserStatus

**user**: Representa al dueño del paquete. Usuario al cual se enviará el paquete con todos los regalos.

**open**: Indica si el paquete está abierto, lo cual significa que aun otros usuarios durante sus compras podrán recibir sus productos como parte de este paquete. Un paquete se cerrará pasado su estado a Open=False cuando el usuario dueño realice cualquier compra.

**message\_mail**: Cuando se cambia el state de un Collpack se envía automáticamente un correo al usuario asociado, el body de este email es indicado en message\_mail

**MODELO COLLPACKPRODUCTS**

**Sergio este comentario en este modelo está mal. Debió decir al final para poner los collpack products en la sección de collpacks.**

class CollpackProducts(models.Model):  
  
 #Este campo contiene al propietario del paquete y no seria necesario ponerlo ya que este modelo  
 #esta relacionado con el Collpack, pero se ha puesto este campo redundante para lograr poner los  
 #collpack products en la tarjeta del producto.  
 user = models.ForeignKey("auth.User", on\_delete=models.CASCADE, null=True, blank=True)  
  
 user\_invited = models.ForeignKey("auth.User", on\_delete=models.CASCADE, null=True, blank=True, related\_name='UserInvited')  
  
 collpack = models.ForeignKey(Collpack, on\_delete=models.CASCADE, null=True, blank=True, related\_name='CollpackProducts')  
 product\_variation = models.ForeignKey(ProductVariation, null=True, blank=True, verbose\_name=\_("Product"))  
 quantity = models.IntegerField(default=0)  
 total\_gift = models.IntegerField(default=0)

Este modelo representa los productos de un paquete.

**user**: El propietario del paquete en el cual se encuentra el producto representado por CollpackProduct. Campo redúndate ya que a través del campo Collpack se puede acceder a él, pero tuve que ponerlo aquí por un detalle técnico que me impedía crear la sección de reportes de collpack en la administración.

**user\_invited**: El user en caso de ser un producto de un usuario que fue invitado por el dueño del paquete.

**collpack**: Collpack al cual pertenece este CollpackProduct.

**product\_variation**: Representa el producto del Ecommerce que se encuentra en el collpack.

**quantity**: Cantidad de veces que el usuario tiene ese producto y el campo **total\_gift** representa cuantas de esas veces se obtuvo como regalo.

En caso del campo user\_invited estar en null significa que el collpackProduct es un producto del dueño del paquete.

**MODELO USER STATIC**

class UserStatic(models.Model, AdminThumbMixin):  
 products\_bought\_by\_friends = models.IntegerField(default=0) #Productos comprados por las amigas  
 partial\_bought\_by\_friends = models.IntegerField(default=0) # Conteo parcial de productos comprados  
 products\_won\_as\_gift = models.IntegerField(default=0) #Productos ganados como regalo  
 total\_points\_won = models.IntegerField(default=0) #Cantidad de puntos ganados  
 points\_won\_for\_ritual\_sharing\_instagram = models.IntegerField(default=0)  
 points\_won\_for\_ritual\_sharing\_youtube = models.IntegerField(default=0)  
 user = models.OneToOneField("auth.User", on\_delete=models.CASCADE, related\_name='UserStatics', null=True)

Este modelo representa estadísticas de los puntos y compras obtenidas por un usuario. Los campos son descriptios a continuación.

**products\_bought\_by\_friends**: Representa la cantidad de productos compradas por las amigas del usuario.

**partial\_bought\_by\_friends:** Representa la cantidad de productos comprados por las amigas que se están contando para saber cuándo el usuario obtiene un regalo. Por ejemplo, si 2 amigas entre ellas compran 7 productos. El contado terminada con un valor de 1 pues 6 productos completaron 2 regalos para el usuario y queda un producto que se acumula para el conteo del próximo regalo.

**products\_won\_as\_gift:** Cantidad de productos ganados como regalos, campo que se puede computar a partir de los dos anteriores pero que seta persistente para en la administración lograr ciertas funcionalidades que no se logra si devuelve el computo del valor, es decir es necesario tenerlo como campo.

total\_points\_won: Total de puntos que ha obtenido el usuario: Sergio mira la siguiente línea, ahí es de donde se lee desde los settings quizás el problema, es que hay que castear a int

points\_won = request.cart.total\_quantity() \* settings.EXPERIMENT\_POINTS\_FOR\_PRODUCT\_SOLD  
user\_static.total\_points\_won += points\_won

Si aún no encuentras el problema fíjate en la línea anterior del checkout ahí es donde está la asignación y es donde algo puede estar pasando, revise rápido pero quizás deba ser int(settings.EXPERIMENT\_POINTS\_FOR\_PRODUCT\_SOLD)

**MODELO USERGAINPRIZE**

class UserGainPrize(models.Model, AdminThumbMixin):  
 user = models.OneToOneField("auth.User", on\_delete=models.CASCADE, related\_name='UserGainPrize', null=True)  
 #Cuenta la cantidad de regalos, a medida que el usuario los escoga  
 #se le ira descontando la cantidad de ellos.  
  
 products\_gift = models.IntegerField(default=0)

Cuando un usuario gana un conjunto de regalos hay que mantener persistente cuantos tiene actualmente, cantidad que se descontara cada vez que el usuario realice una compra y en el carrito existan productos marcados por el cómo regalos.

Notar que el usuario puede ganar regalos por las amigas/referidos que compran, pero también si el mismo hace una **compra** durante la cual **compra** más de 3 productos. Los únicos que están exentos de ganar regalos cuando ellos mismos compran más de 3 son los usuarios que no tienen status es decir que son invitados y aun no están registrados en el sistema o el operador les quito su estado.

**DOCUMENTACION DEL VPS**

Como parte del material de entrega se distribuyen varios videos que permite ver casos de uso del VPS. A continuación, se explica el contenido de cada video:

**Actualización VPS.avi**: Muestra cómo se procede para actualizar el sistema beautydates. Los pasos para actualizar cualquier sistema Django son los mismos. En el ejemplo/video se han ejecutado todos los comandos posibles, claro está que no es necesario ejecutar migraciones si los cambios subidos al repo no implican ejecutar ninguna migración.

**Actualización Git + VPS.avi**: Muestra la actualización del repo del sistema beautydates usando el cliente de git smartgit.

**Configuraciones Django VPS.avi**: Muestra las configuraciones en el VPS que permiten hacer los despliegues de sistemas Django. Complementario a estos videos son los documentos en la carpeta Django + Nginx. Importante cuando se despliega un nuevo sistema en el VPS en el caso de Django es necesario hacer un soft link al file de configuración que queda bajo el directorio /etc/nginx/sites-available, revisar la documentación **Serving multiple Django applications with Nginx and Gunicorn - Micha\_\_ Karzy\_\_ski.html** sección Enable the virtual servers and restart Nginx:

**Bitbucket + Laravel Config VPS + Domain All Sites.avi**: Muestra varios temas, la interfaz de bitbucket clonado de repositorios, los dominios actuales de todos los sistemas y los distintos files para hacer restart de los servicios de Django.

En la carpeta VPS Documentacion/Configuraciones de Ejemplo se encuentran las configuraciones para desplegar el sistema beautydates, las cuales fueron copiadas online como se ve en el video **Configuraciones Django VPS.avi** del cual se infiere la ubicación de estas configuraciones.

Importante cuando se despliega un nuevo sistema Django.

1. Hacer el soft link.
2. Restart del server “service nginx restart” o hacer un reload para solo tener que leer las configuraciones.