Django로 쇼핑몰 만들자

정경업(파이) 2016. 8. 13

파이라고 불러주세요.

웹 개발을 합니다.

Django를 주로 씁니다.

~~풀 스텍...?~~

간단한 이력

- **현재 백수** - 퇴사후 지금까지
- **스마트스터디** 소프트웨어 엔지니어 - *3년 반*
  - store.pinkfong.co.kr
  - alpacacomics.co.kr
- **게임동아** 개발 팀장 - *3년*
  - it.donga.com
  - game.donga.com
- 게임 팬사이트 제작 - *15년 전*

기술 경험

- back: django(python) / classic asp
- front: html5 / css(less) / javascript(jquery)
- design: image editing / responsive / material / brand logo
- deploy: aws / docker / nginx / uwsgi / apache
- os : macOS / linux(ubuntu) / windows server

연락

정경업(파이)

perhapsspy@gmail.com

1. 어쩌다보니

쇼핑몰을 만들었습니다.

2. 만들다보니

경험자를 찾는데 별로 없어요.

3. 만들고나니

이거 발표할만 한듯..?!

이런 이야기를 해보겠습니다.

- 쇼핑몰에 대한 대략적인 이해
- 이미지, 제품, 주문, 장바구니, 결제, 관리자
- 테스트 작성해가며 복잡한 적립금 구현

할 말은 많지만 시간이 없어요.
OST에서 자세한 질문 받겠습니다.

필요 -> 공부 -> 적용

~~절실하면 파이썬이 도와줌~~

1. 필요

쇼핑몰!!!

최초 요청

일단 **어떻게든 물건을 팔면** 됩니다.

현실

이것도 되고 저것도 되고 **될건 다 되야...**

개발자는?

파이

+ 신입 한명

왜 그랬을까

2. 공부

근데... 쇼핑몰이 뭐였지...?!

만들려고 보면 막막...

최소 필요로 하는 것

- 제품
- 장바구니
- 주문
- 결제
- 위 항목들 관리 기능

이 정도는 더 있어야

- 재고
- 회원
- 배송 확인
- 이메일 알림
- 정산

이런 것도 더 ~~불어...?~~

- 마케팅용 페이지
- 할인 판매
- 매출 보고서 및 통계 (자체 / Google analytics)
- API (물류 관리 등 외부 서비스)
- 적립금 ~~(살려줘)~~
- ...

“ 나는 이것을 경이로운 방법으로 증명하였으나, 책의 여백이 충분하지 않아 옮기지는 않는다.

피에르 드 페르마 ”

“ 나는 이것을 대강 다 만들긴 했으나, 발표의 시간이 충분하지 않아 옮기지는 않는다.

아무말이나 하는 파이 ”

3. 적용

쇼핑몰 부분만 만들었습니다.

물류 관리는 창고 계약에 따라,
운영 규모에 따라 천차만별

제품을 구현해봅시다.

제품의 기본은 이미지

이미지 다루는 방법은 많다.

가장 현재 상황에 효과적인 것은?

이미지 저장과 서빙(CDN)

S3(SimpleStorageService)를 저장소로 사용

```
# settings.py
DEFAULT_FILE_STORAGE = \
  'storages.backends.s3boto.S3BotoStorage'   # 쉽네.
```

CF(CloudFront)를 통한 CDN 서비스

```
{% load image_url %} <!-- custom template tag로 할만함 -->
<img alt="{{ image.alt }}"
     src="{{ image|image_cf_url }}"/>
```

그렇다면 섬네일은?

django-versatileimagefield

~~버써털..? 버쓰타일?~~ ImageField를 대체; 유연함, 직관적, 쉬운 확장

요청시 바로 생성

미리 준비도 가능

```
# settings.py
VERSATILEIMAGEFIELD_RENDITION_KEY_SETS = {
    'product_image': [
        ('order_list', 'crop__50x50'),
        ('detail_list', 'crop__480x480'),
    ],
}
```

```
# product/models.py
@receiver(models.signals.post_save, sender=ProductImage)
def warm_product_images(sender, instance, **kwargs):
    warmer = VersatileImageFieldWarmer(
        instance_or_queryset=instance,
        rendition_key_set='product_image',
        image_attr='image')
    num_created, failed_to_create = warmer.warm()
```

이미지 모델 재사용

```python
# 공용 추상 모델
class ImageBaseModel(models.Model):
    class Meta:
        abstract = True
    image = VersatileImageField(upload_to=image_upload_to,
        ppoi_field='ppoi', blank=False,
        width_field='width', height_field='height')
    ppoi = PPOIField()
    height = models.PositiveIntegerField(blank=True, null=
    width = models.PositiveIntegerField(blank=True, null=T
```

```python
# 필요한 곳에서 추가
class ProductImage(ImageBaseModel, OrderedModel, TimeStamp
class ProductPresentation(ImageBaseModel, SortableMixin, Y
```

추상 모델 (abstract model)

~~모델 상속 지옥의 시작~~

- \+ 같은 일 두번 안함
- \- 코드 읽기 힘듬
- \- 구성이 어려움

이미지가 준비되었으니
이제 정말 제품을 봅시다.

제품은

사용자 편의 / 마케팅으로

표현하고자 하는

정보가 많다.

두가지 고민

어떻게 판매할 것인가 - 운영 - 마케팅

어떻게 보여줄 것인가 - UX - 디자인

정답이 없어요

상황에 따라 유연하게 대응

구조를 잘 짜야

제품 모델

- 제품: 간단 설명, 부가정보, 관계, 판매, 상태 표현 등
  - 가격: 정가, 할인가, 면세, 배송비, 할부 등
- 제품 설명: 이미지 - 제목 - 설명 구조
  - 이미지 모델을 상속 받아 공유(개발 편의)
  - 다시 만든다면? Rich text로 만들겠음

자주 상속 받는 가격 모델

```
class PriceModel(models.Model):
    class Meta:
        abstract = True
    # 제품 가격 정보
    price = models.PositiveIntegerField(u'정가', default=0)
    discount_price = models.PositiveIntegerField(u'할인금액'
    tax_free = models.BooleanField(u'면세', default=False)
    delivery_charge = models.PositiveSmallIntegerField(u'배
    # 제품/상품 구분
    is_goods = models.BooleanField(u'상품', default=False,
    # 결제 방식
    interest_free_month = models.PositiveSmallIntegerField
        u'무이자 할부 월', null=True, blank=True)
    # 적립금 정보
    point_amount = models.IntegerField(u'적립될 포인트', defau
    point_expire_months = models.IntegerField(u'적립될 포인트
```

제품 모델 코드 상속 구조

```
class Product(ActiveModel, PriceModel, PreSalesModel,
              YAMLLoadMixin, EditableTimeStampedModel,
              CacheDeleteModel):
    class Meta:
        verbose_name = u'제품'
        verbose_name_plural = verbose_name

    objects = models.Manager()
    live_objects = LiveProductManager()
    # 제품 관계, 설명, 부가정보, 판매 관리, 판촉 문구 노출 관리 등

class Category(TimeStampedModel, CacheDeleteModel):
class ProductImage(ImageBaseModel, OrderedModel,
                   TimeStampedModel):
class ProductPresentation(ImageBaseModel, SortableMixin,
                          YAMLLoadMixin, TimeStampedModel)
class ProductMovie(TimeStampedModel):
```

추상 모델과 믹스인

추상 모델(Abstract Model)

- field 포함
- TimeStampedModel : created, updated 포함

믹스인(Mixin)

- field 없이 내용만 있을때
- SortableMixin : 정렬을 위한 기능만 있음

제품이 대강 준비 되었다면

사봅시다.

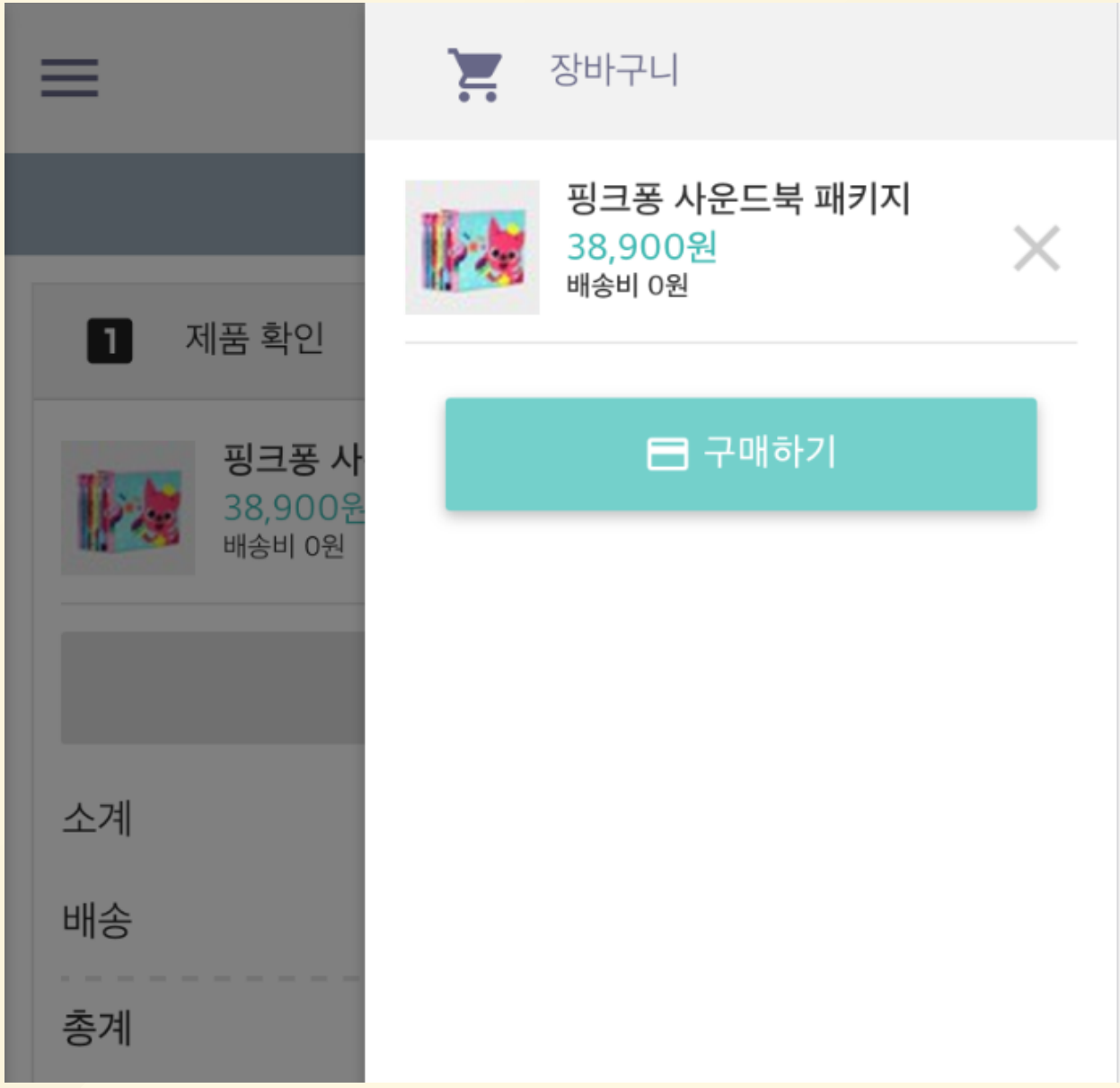
일단 장바구니에 담아야죠.

장바구니 구현의 귀찮음

- 어디서나 보여야
- 주문 수량에 따라 가격을 계산
- 주문 총 금액에 따라 무료 배송 처리
- 비회원도 사용 가능해야함

어떻게든...

- 어디에서나 사용 가능하게 Ajax로 구현
- 편집 View에서 가격 계산도 같이함
- 주문서에도 장바구니 편집 View를 재활용
- 쿠키 기반 장바구니 연동 추가



장바구니와 주문시 제품 확인 부분은 같은 코드를 씀

장바구니 CBV 코드 일부

```
class GoodsInCartView(AjaxChangeTemplateMixin, ListView):
    model = GoodsInCart
    template_name = 'shoppingcart/_list.html'
    ajax_template_name = 'shoppingcart/_list.html'

    def get_queryset(self):
        # ...
    def update_cart(self, request, value, shopping_cart):
        # ...
    def post(self, request, *args, **kwargs):
        # ...
    def get_context_data(self, **kwargs):
        # ...
```

CBV를 잘쓰면 View가 편합니다.

대부분의 일은 모델 설계에서 끝납니다.

주문을 해봅시다.

판매(물류)

1. 상품 고르기
2. 장바구니에 담기
3. 주문서 작성
4. 결제
5. 제품 준비 및 배송 / 혹은 결제 취소
6. 배송 완료

~~오예~~

환불/교환(역물류)

1. 배송 완료한 물건에 문제 발생 혹은 변심
2. 환불 / 교환 신청
3. 이후 물건 상태에 따라 상당히 복잡한 일들이 발생
4. 대부분 사람이 해결하고 기록을 남김

~~지옥, 혹은 불지옥~~

물류와 역물류

거래 기준은 주문서

주문서와 역주문서를 나눔

둘의 성격과 처리 방식이 달라 분리

정산은 합쳐서 처리

주문 과정에 따라 상태는 변함

주문 -> 결제 진행 -> 준비 -> 출고 -> 배송 완료

마지막 상태 유지하고

과정은 기록이 필요

주문(물류 방향) 모델들

주문 상태 추상 모델(OrderStatusModel)

- 주문 상태를 주문서와 기록 모델에서 공유

주문서(Order)

주문한 제품(OrderdProduct)

- 주문한 시점의 제품 가격 고정

주문서 상태 기록 모델(OrderStatusLog)

주문서 상태 추상 모델

```
class OrderStatusModel(models.Model):
    class Meta:
        abstract = True
    STATUS_CHOICES = [
        ('processing', u'주문 진행 중(결제 필요)'),   # 1
        ('cancelled', u'주문 취소됨'),   # 1-1
        ('payment-waiting', u'결제 대기 중'),   # 2
        ('payment-fail', u'결제 실패'),   # 3-2
        ('paid', u'결제 완료'),   # 3-1
        ('paid-cancelled', u'결제 취소됨'),   # 4-2
        ('preparing', u'제품 준비 중'),   # 4-1
        ('delivery-waiting', u'출고됨'),   # 5
        ('delivered', u'배송완료'),   # 6
    ]
    status = models.CharField(u'상태', default='processing'
        max_length=50, choices=STATUS_CHOICES)
```

주문서 모델

```
class Order(UpdateParamsMixin,
            ActiveModel, OrderStatusModel,
            AddressMixin, PriceModel,
            TimeStampedModel, CacheDeleteModel):
    # 항목이 많아 생략
    # 주문자, 배송지, 적립금 등
```

- OrderStatusModel: 기록 모델과 공유
- AddressMixin: 회원 주소록 관련 기능 공유
- PriceModel: 제품과 가격 모델 공유

주문한 제품 모델

제품의 가격 정보를 결제 완료 시점으로 굳힘

```
class OrderedProduct(AdminOrderDisplayMixin, PriceModel, T
    class Meta:
        verbose_name = u'주문한 제품'
        verbose_name_plural = verbose_name
    order = models.ForeignKey(Order, verbose_name=u'주문서')
    product = models.ForeignKey(Product,
                                verbose_name=u'제품')
    quantity = models.PositiveIntegerField(u'수량', default
    # ...
```

주문서 상태 기록 모델

주문 진행 상태 변화만 기록함

```
class OrderStatusLog(AdminOrderDisplayMixin,
                     OrderStatusModel, TimeStampedModel):
    class Meta:
        verbose_name = u'주문서 상태 기록'
        verbose_name_plural = verbose_name

    order = models.ForeignKey('Order')
    # ...
```

역물류도 비슷한 구조

설명은 생략합니다.

역주문서, 역주문한 제품, 역주문서 상태 기록

~~시간이 없을 듯~~

일단 돌아는가지만...

다시 만든다면
'주문서 상태 기록'을 '주문서 기록' 모델로 바꿔서
중복 저장하는 식으로 만드는게 좋을 듯.

* 정산 때문에 어차피 역정규화함
* 이후 적립금 구현은 그렇게 함.

주문서 작성은?

- 장바구니에서 언급했듯, 편집 View를 Ajax로 공유해서 씀
- OrderForm에서 Save시
  장바구니 내용을 엮어서 주문서를 생성
- 주소는 다음 우편번호 서비스 API를 사용
- 주소록을 기록해 놨다가 손쉽게 읽어올 수 있게 함
- 장바구니 내역과 주문서의 주문한 제품을를 동기화
  * 코드가 지저분하지만 작동은 함

order/forms.py 코드 일부

```
class OrderForm(PhoneNumberCleanKrMixin,
                PostCodeCleanMixin, NameCleanMixin,
                forms.ModelForm):
  class Meta:
    model = Order
    fields = [
      'billing_name', 'billing_phone', 'billing_email',
      'name', 'phone', 'zonecode', 'postcode',
      'address', 'address_old', 'address_detail',
      'comment', 'point_use', 'payment_method'
    ]
    widgets = {
      'comment': forms.Textarea(),
      'phone': PhoneNumberWidget(),
      'billing_phone': PhoneNumberWidget(),
      'payment_method': forms.RadioSelect(),
    }
```

```python
    # 이어서
    def save(self):
        order = self.make_order()
        # ... 회원 정보관련 업데이트 처리
        return order

    def make_order(self):
        # 주문서 생성
        # 장바구니, 주문한 제품과 동기화
        # 주문서 정보 업데이트
        return order
```

폼에서 입력시 주문 관련 로직을 거의 처리함
유효성 검사하는 Mixin 재사용

결제를 해보자

보통은 지옥이나...

~~어느 PG사든 연동을 직접 해보세요.~~

지옥을 정복한 아임포트

다양한 PG사 연동을 Restful API로 서비스

개발자 고통 감소

필요한 기능 대응 빠름

결제 모델은 주문서와 분리

주문서(Order) - 결제(Payment)

영수증 모아놓듯
하나의 주문서에 다른 결제가 여럿 가능
(결제 완료 후 취소시 등)

결제 로직

1. 주문 작성
2. 결제 시작(아임포트 javascript 호출)
3. 사용자가 PG사와 결제 진행(PG사 Active-X 등)
4. 결제된 내용 아임포트쪽에 확인(서버 API)
5. 쇼핑몰에서 결제 완료로 처리

결제 완료 확인 로직

1. 아임포트에서 넘겨준 URL이 맞는지 확인
2. 주문서 읽기
3. 아임포트에서 결제 읽기
4. 주문서와 결제 내역 확인
5. 재고 다시 확인
6. 주문서와 결제를 완료로 처리

위 과정 중 문제 발생시 결제 취소
주문서도 취소로 종결

결제 완료 확인시 예외 처리 코드 일부

```
def payment_view(request):
  # ...
  # 결제 금액이 맞는지 확인
  if not iamport.is_paid(order.payment_price,
                         response=iamport_response):
    order.update_status('processing')
    messages.error(request,
                   u'결제 금액이 맞지 않아 결제에 실패하였습니다.')
    return payment_cancel_redirect(request, merchant_uid)
  # 재고가 모자른 상황 예외 처리
  if order.not_enough_stock_ordered_products():
    order.update_status('processing')
    messages.error(request,
                   u'주문 중 재고가 부족하여 결제가 되지 않았습니다.'
    return payment_cancel_redirect(request, merchant_uid,
                                   'shopping-cart')
  # ...
```

가장 많이 겪는 문제

돈만 나가고 **결제 완료 안됨**

- 결제 완료 확인 로직을 중복 실행 가능하게 만듬
- 관리자 페이지에서 확인 가능

오픈 소스로 공개

I'mport; REST Client

```
# pypi에 등록도 했어요.
pip install iamport-rest-client
```

기여도 받음

- 파이썬3 지원, 테스트
- 비인증결제
- 부분취소

결제가 잘되었다면 창고는

1. 물류 창고에서 주문 내역 받기(엑셀)
2. 주문서 개별 출력
3. 주문서 보며 박스 포장
4. 택배 운송장 붙임
5. 송장번호 포함된 주문 내역 쇼핑몰에 올리기(엑셀)
6. 주문서 상태 배송 중으로 변경

관리자 페이지는 어떻게?

django admin 개조

- 관리자 기능을 모두 만들기엔 인력 부족
- 기본 그룹 기능으로 권한 분리
- 추가 페이지와 javascript로 얹어서 편의기능 구현

가장 많이 마개조된 모델은 제품

- 관계된 모델 admin inlines 활용
- 정리를 위해 fieldset 지정
- 제품 설명 미리보기 구현
  - 이미지 사용으로 모델로 구현됨
  - 한 눈에 보기 어려움
  - 프론트 템플릿 사용하여 모달 처리
    - Plate.js - Django template과 유사

product/admin.py 코드 일부

```
list_display = ['id', 'active', 'is_home', 'sales_state',
    parent_product_display, name_display, current_stock,
    sales_price_display, discount_rate_display,
    price_display, discount_price_display,
    delivery_charge_display, 'point_amount',
    'tax_free', 'is_goods',
    'created']
list_editable = ['created', 'is_home']
list_filter = ['sales_state', 'active', 'is_goods', 'tax_f
search_fields = ['name', 'model_name', 'slug', 'subtitle']
date_hierarchy = 'created'
inlines = [
    ProductImageInline, ProductMovieInline,
    ProductPresentationInline
]
raw_id_fields = ['parent', ]
```

가격 계산 스크립트(할인율 % 등)

재고 확인 / 입력 스크립트

~~**Django 발표라 javascript 코드는 생략합니다.**~~

창고와의 연동

- 창고에서 사람이 직접
- 주문서 엑셀 다운로드 / 업로드 페이지 구성
  - 상단 메뉴에 우겨 넣음

  - get_urls 덮어 씌움
  - 엑셀 내용 미리보기 구성

테스트를 작성하며

복잡한 적립금 구현하기

적립금 구현 이야기

- 처음엔 가볍게 생각해서 금방 만듬
- 유효기간이 잡힌 정도
- Point, PointLog, PointAction 3가지 구조로 구현
- 회의 때마다 조건이 쌓여감(~~안되겠어 어떻게든 하지 않으면~~)
- 테스트를 짜며 만들기로 함

* Action: 적립, 사용, 취소, 회수
* 페이스북에 글 써서 한탄한적이 있음

왜 복잡할까

- 유효기간, 금액이 적립 방식에 따라 유동적(구매/이벤트)
- 환불과 엮임(역물류)
- 사용시 VAT 계산과 정산이 되야함
- 고객서비스로 유연한 대응이 필요
- ...

~~많은 삽질 끝에~~

결과적으로 짜게 된 테스트 중

Action에 관한 테스트 시나리오

유닛 테스트

- 적립(+) : 중복 방지, 적립할 금액 산출, 적립 완료 여부
- 사용(-) : 사용 로그 생성, 주문 비율 계산 후 개별 VAT
- 취소(+) : 취소 로그 생성
- 회수(-) : 회수 로그 생성

  1. 해당 주문에 사용된 적립금을 취소
  2. 가지고 있는 적립금에서 회수
  3. 미회수 적립금 역주문서에 기록
     * 미회수 적립금: 주문 적립금과 회수금을 비교

시나리오 테스트

1. 주문 - 결제 완료 - 배송 완료 -
    a. 적립
    b. 적립 - 주문 환불 - 적립금 회수
2. 주문 - 사용 - 결제 완료 - 주문 취소 - 결제 환불 - 사용 취소
3. 주문 - 사용 - 결제 완료 - 배송 완료 -
    a. 적립
    b. 적립 - 주문 환불 - 회수 - 사용 취소
    c. 적립 - 적립금을 다른 주문에 사용 - 주문 환불 - 회수
    * '적립금 사용 취소'는 cs에서 유동적으로 하기로 바꿈

~~너무 많아~~

만들 테스트가 많은데
Django 기본 Test는 느립니다.

- 테스트가 쉽고 빠르지 않으면 고통속에 안짜게됨
- DB 생성, 마이그레이션 절차가 특히 병목
- gist: Django에서 Test 쉽고 빠르게 하기
  - py.test
  - pytest-django
  - model_mommy
    * 더미데이터를 선행 절차 없이 있는 것처럼 다룸

test_point_action.py 코드 일부

```
from model_mommy.mommy import make as mm

class TestPointAction(TestCase):
    def setUp(self):
        self.user = mm(User, username='py',
                       email='py@test.com')
        self.pa = PointAction(user=self.user)
        # ...

    def test_use_u1(self):
        self.pa.accumulate(300000)
        self.assertRaises(OverflowPointError, self.pa.use,
                          order=self.order_a,
                          amount=30000)
    # ...
```

테스트와 함께 만든 모델

```python
class BasePointModel(UpdateParamsMixin, TimeStampedModel):
    class Meta:
        abstract = True
        ordering = ['-created']
        # 소유자, 이유, 내용, 수명, 관계(근거)

class Point(BasePointModel):  # 현재 적립되어 있는 적립금
    def save(self, *args, **kwargs):
        # 최초 생성시 적립금 적립 기록 생성

class PointLog(BasePointModel):  # 적립금 기록
    point = models.ForeignKey(Point, null=True, blank=True
```

point/action.py 코드(함수만)

```
class PointAction(object):
    def __init__(self, user):
        self.user = user

    def accumulate(self, amount, **kwargs):
    def accumulate_by_order(self, order):
    def use(self, order, amount):
    def cancel(self, order):
    def withdraw(self, reverse_order):

    def points(self, expire=False):
    def logs(self, order, category=None):
    def amount(self):
    def will_expire_amount(self):
    def debt_amount(reverse_order):
    def withdraw_point(self, reverse_order):
    def debt_point(self, reverse_order):
```

적립금 사용 View 테스트

django-test-plus 사용

response 확인, 로그인 유지 등을 좀 더 쉽게할 수 있음

test_point_view.py 코드 일부

```
from model_mommy.mommy import make as mm
from test_plus.test import TestCase

class PointPaymentView(TestCase):
  def setUp(self):
    self.user = mm(User, is_active=True)
    self.user.set_password('password')

  def test_payment_view_s1(self):
    self.assertLoginRequired('point-payment')
    with self.login(username=self.user.username, password=
      response = self.get('point-payment',
      data={'order_id': self.order_a.id})
      self.get('shopping-cart')
      self.response_302(response)
      message = unicode(self.get_context('messages').__ite
      assert message == u'결제가 완료 되었습니다.'
```

끝

질의 응답 받겠습니다.

발표 자료 제작에 많은 도움을 주신

석우징님 감사합니다.

약간의 부록이 있어요

부록

- 이메일
- 배송 확인
- 재고
- 리포트
- 프론트엔드
- 슬라이드 작성 툴

주문이 진행 내용을 사용자에게 알리자

제일 기본(만만한)이 이메일 전송

이메일을 보내는 타이밍

1. 주문 결제 완료시
2. 배송 시작시
3. 회원 가입시
4. 문의에 대한 답변

Django에서 이메일 보내기

- Django는 이메일 모듈이 잘 되어 있음
- 이미지는 최소화 (로고만)
- Backend: 아마존 SES (django-ses)

이메일 디자인의 문제

- 최대한 단순한 디자인 유지
- 이메일 클라이언트에서 지원하는 CSS는 제한적
  - inline css만 작동하는 Gmail 등.
  - django-inlinecss로 해결
- 그래도 모르니 Rich text와 Plain text를 섞어서 보냄
  - 이메일 하나당 Template 2개(html, txt)

보냈으면 확인해야지

배송 확인을 해보자

배송 확인

- 정산시 매출 기준이 배송 확인된 주문서
- 로젠 API 배송 확인 개발해서 공개
  - 메뉴얼 부실
  - SOAP을 사용

분리된 별도의 서비스 운영

- django-celery 사용
- 1시간 마다 확인
- DRF로 주문서 API를 구현

1. 쇼핑몰에서 배송 확인 대상 주문서 목록 받음
2. 로젠 API를 통해 배송 완료 여부 확인(날짜 포함)
3. 완료된 내역을 쇼핑몰로 보냄

재고 모델

- 재고 : 현재 재고 - 추가만 가능
- 재고 기록 : 판매, 추가, 취소 등의 기록 모두 남김

1. 재고 입력
2. 결제시 재고 유무 확인
3. 결제 완료시 재고를 감소
4. 결제 취소시 재고 증가
5. 환불과는 관련 없음

```
class ProductStock(TimeStampedModel):
    product = models.ForeignKey(Product, verbose_name=u'제품
    quantity = models.IntegerField(u'수량', default=1)

    def update_quantity(self, variation, order,
                        reverse_order=None):
        result = self.__class__.objects.filter(
            id=self.id).update(
                quantity=F('quantity') + variation)
        ProductStockLog.objects.create(
            user=order.user, order=order, stock=self,
            quantity=variation, reverse_order=reverse_orde
        return result

    def save(self, *args, **kwargs):
        new = True if not self.id else False
        super(ProductStock, self).save(*args, **kwargs)
        if new:
            ProductStockLog.objects.create(
                stock=self, quantity=self.quantity)
```

```
class ProductStockLog(TimeStampedModel):
    user = models.ForeignKey(User, null=True, blank=True)
    order = models.ForeignKey(Order, verbose_name=u'주문서',
    reverse_order = models.ForeignKey(
        ReverseOrder, verbose_name=u'역주문서', null=True, bl
    stock = models.ForeignKey(ProductStock, verbose_name=u
    quantity = models.IntegerField(u'변동 수량', default=1)
```

- 재고 변동은 ProductStock 기준
- 판매시 update_quantity, 재고 발생시 save에 Log를 쌓게끔 구성됨
- Django admin에서 ProductStock을 추가하는 것만으로 재고 관리 가능

리포트

- 정산, 회원/주문 모니터링 목적
- admin-lte 프레임워크로 프론트 구성
- 최대한 단순한 작업으로 리포트를 추가하기 좋게 만듬
  - excel mixin 만들어서 재활용
- Django aggregation 활용
  - DB 부하 최소화 노력

Front-end

예..예쁘게..?

~~가볍게~~

중요한 것

검색/소셜 친화적

모바일 우선

반응형 디자인

검색/소셜 친화적

여러가지 규약을 챙겨야합니다.

웹 표준 준수는 기본

챙겨할 마크업

<meta>

title, keyword, og, twitter

<link>

icon, shortcut, apple, android, ms

<a>

rel="nofollow"

template은 다소 지옥

```
<!--base.html-->
<meta property="og:title"
 content="{% block og_title %}쇼핑몰 이름{% endblock %}"/>
...
<!--product/detail.html-->
{% extends 'base.html' %}
{% block title %}크고 아름다운 인형{% endblock %}
{% block meta_title %}크고 아름다운 인형{% endblock %}
{% block og_title %}크고 아름다운 인형{% endblock %}
{% block tw_title %}크고 아름다운 인형{% endblock %}
{% block meta_description %}설명 귀찮아...{% endblock %}
{% block og_description %}설명 귀찮아...{% endblock %}
{% block tw_description %}설명 귀찮아...{% endblock %}
...
```

어떻게든 ~~검색/소셜 친화적~~

모바일 우선 + 반응형 디자인

Materialize

머티리얼 디자인 기반 반응형 프론트엔드 프레임워크

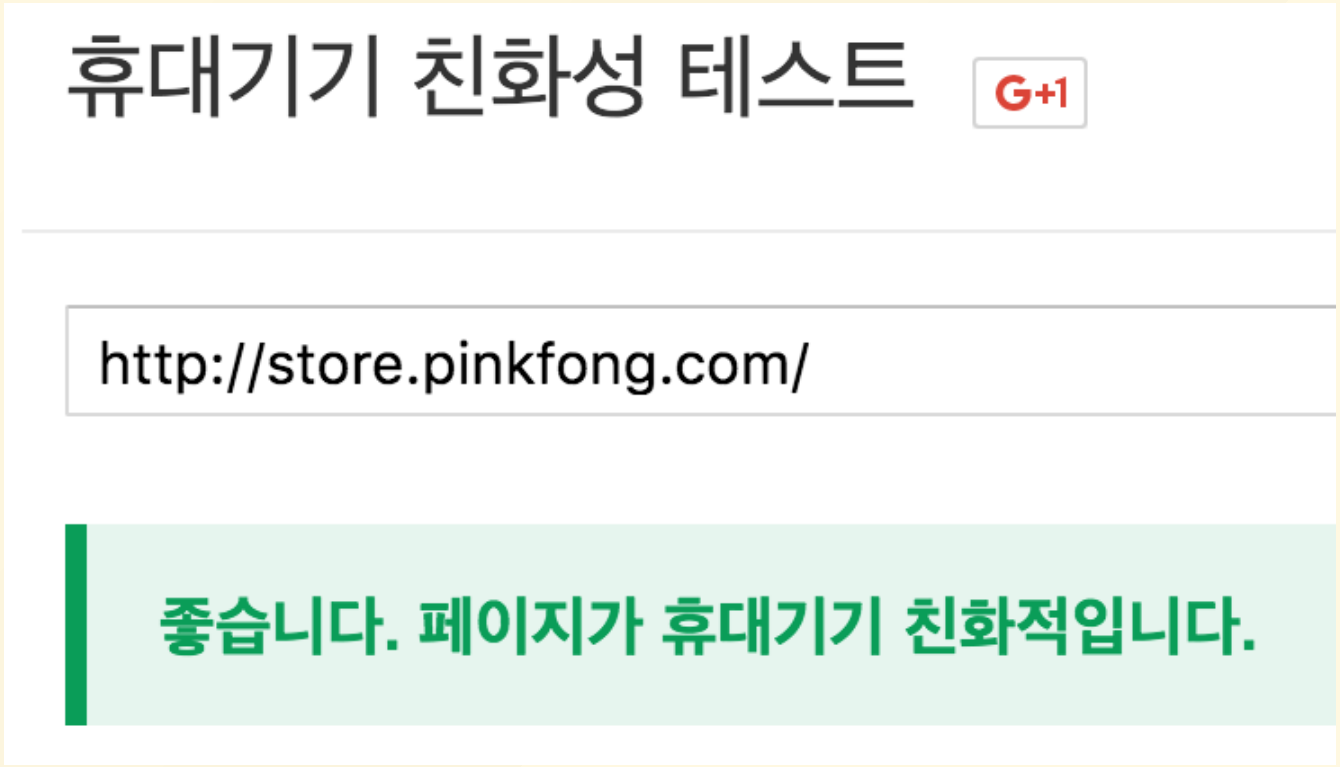
~~마침 좋아보이는 물건을 발견~~

Materialize 기반으로 작업

- 구글 머티리얼 디자인은 모바일 앱 위주로 큰 화면은 어색
- 추가적인 CSS 덮어 씌움
- 동적인 부분은 jquery로 충분(장바구니 정도)
- css를 그냥 쓰면 고통. lesscss 씀

~~**모바일 우선**~~

~~**반응형 디자인**~~



“ 자동으로 만들어주는 서비스가 많으니 찾아서 이용합시다.
접근성을 테스트하는 서비스도 많이 있습니다.
구글, 애플, 트위터, 마소 개발자 페이지 자주 갑시다.
세상 쉽게 살고픈 파이 ”

슬라이드 작성 툴

Marp 0.0.8

Markdown Presentation Writer

감사합니다.