Clean Architecture in Python

Sebastian Buczyński

@ PyKonik Tech Talks #36
SOFTWARE ENGINEERING?

PIP INSTALL
Clean Architecture

1. Independence of frameworks
2. Testability
3. Independence of UI
4. Independence of database
Clean Architecture

Putting customer's concerns in the first place
Project: Auctions online
User stories

- As a bidder I want to make a bid to win an auction
- As a bidder I want to be notified by e-mail when my offer is a winning one
- As an administrator I want to be able to withdraw a bid
Django + Rest Framework!
class Auction(models.Model):
    title = models.CharField(...)
    initial_price = models.DecimalField(...)
    current_price = models.DecimalField(...)

class Bid(models.Model):
    amount = models.DecimalField(...)
    bidder = models.ForeignKey(...)
    auction = models.ForeignKey(Auction, on_delete=PROTECT)
User stories

- As a bidder I want to make a bid to win an auction ✔
- As a bidder I want to be notified by e-mail when my offer is a winning one ✔
- As an administrator I want to be able to withdraw a bid
def save_related(self, request, form, formsets, *args, **kwargs):
    ids_of_deleted_bids = self._get_ids_of_deleted_bids(formsets)
    bids_to_withdraw = Bid.objects.filter(pk__in=ids_of_deleted_bids)

    auction = form.instance
    old_winners = set(auction.winners)
    auction.withdraw_bids(bids_to_withdraw)
    new_winners = set(auction.winners)

    self._notify_winners(new_winners - old_winners)

    super().save_related(request, _form, formsets, *args, **kwarg
def save_related(self, request, form, formsets, *args, **kwargs):
    ids_of_deleted_bids = self._get_ids_of_deleted_bids(formsets)
    bids_to_withdraw = Bid.objects.filter(pk__in=ids_of_deleted_bids)

    auction = form.instance
    old_winners = set(auction.winners)
    auction.withdraw_bids(bids_to_withdraw)
    new_winners = set(auction.winners)

    self._notify_winners(new_winners - old_winners)
    super().save_related(request, _form, formsets, *args, **kwargs)
def save_related(self, request, form, formsets, *args, **kwargs):
    ids_of_deleted_bids = self._get_ids_of_deleted_bids(formsets)
    bids_to_withdraw = Bid.objects.filter(pk__in=ids_of_deleted_bids)

    auction = form.instance
    old_winners = set(auction.winners)
    auction.withdraw_bids(bids_to_withdraw)
    new_winners = set(auction.winners)

    self._notify_winners(new_winners - old_winners)

    super().save_related(request, _form, formsets, *args, **kwargs)
class WithdrawingBid:
    def withdraw_bids(self, auction_id, bids_ids):
        auction = Auction.objects.get(pk=auction_id)
        bids_to_withdraw = Bid.objects.filter(pk__in=bids_ids)

        old_winners = set(auction.winners)
        auction.withdraw_bids(bids_to_withdraw)
        new_winners = set(auction.winners)

        self._notify_winners(new_winners - old_winners)
UseCase - Orchestrates a particular process
What about tests?!

Business logic is coupled with a framework, so are tests...
Testing through views

from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):
        User.objects.create(...)

        response = self.client.get('/dashboard/)

        self.assertRedirects(response, '/accounts/login/')
How a textbook example looks like?

class MyTest(unittest.TestCase):
    def test_add(self):
        expected = 7

        actual = add(3, 4)

        self.assertEqual(actual, expected)

No side effects and dependencies makes code easier to test
class WithdrawingBidUseCase:
    def withdraw_bids(self, auction_id, bids_ids):
        auction = Auction.objects.get(pk=auction_id)
        bids_to_withdraw = Bid.objects.filter(
            pk__in=bids_ids)

        old_winners = set(auction.winners)
        auction.withdraw_bids(bids_to_withdraw)
        new_winners = set(auction.winners)

        self._notify_winners(new_winners - old_winners)
Getting rid of dependencies: hide them

class WithdrawingBidUseCase:
    def withdraw_bids(self, auction_id, bids_ids):
        auction = self.auctions_repository.get(auction_id)
        bids = self.bids_repository.get_by_ids(bids_ids)

        old_winners = set(auction.winners)
        auction.withdraw_bids(bids)
        new_winners = set(auction.winners)

        self.auctions_repository.save(auction)
        for bid in bids:
            self.bids_repository.save(bid)

        self._notify_winners(new_winners - old_winners)
class WithdrawingBidUseCase:
    def withdraw_bids(self, auction_id, bids_ids):
        auction = self.auctions_repository.get(auction_id)
        bids = self.bids_repository.get_by_ids(bids_ids)

        old_winners = set(auction.winners)
        auction.withdraw_bids(bids)
        new_winners = set(auction.winners)

        self.auctions_repository.save(auction)
        for bid in bids:
            self.bids_repository.save(bid)
        self._notify_winners(new_winners - old_winners)
Clean Arch - building block #2

class AuctionsRepo(metaclass=ABCMeta):

    @abstractmethod
    def get(self, auction_id):
        pass

    @abstractmethod
    def save(self, auction):
        pass
Clean Arch - building block #3

class DjangoAuctionsRepo(AuctionsRepo):
    
def get(self, auction_id):
        return Auction.objects.get(pk=auction_id)

Interface Adapter / Port Adapter
class WithdrawingBidUseCase:
    def __init__(self, auctions_repository: AuctionsRepo):
        self.auctions_repository = auctions_repository

django_adapter = DjangoAuctionsRepo()
withdrawing_bid_uc = WithdrawingBidUseCase(django_adapter)
import inject

def configure_inject(binder: inject.Binder):
    binder.bind(AuctionsRepo, DjangoAuctionsRepo())

inject.configure_once(configure_inject)

class WithdrawingBidUseCase:
    auctions_repo: AuctionsRepo = inject.attr(AuctionsRepo)
Benefits from another layer

- It is easier to reason about logic
- It is possible to write TRUE unit tests
- Work can be parallelized
- Decision making can be delayed
Our logic is still coupled to a database!

class WithdrawingBidUseCase:
    def withdraw_bids(self, auction_id, bids_ids):
        auction = self.auctions_repository.get(auction_id)
        bids = self.bids_repository.get_by_ids(bids_ids)

        old_winners = set(auction.winners)
        auction.withdraw_bids(bids)
        new_winners = set(auction.winners)

        self.auctions_repository.save(auction)
        for bid in bids:
            self.bids_repository.save(bid)

        self._notify_winners(new_winners - old_winners)
class Auction:
    def __init__(self, id: int, title: str, bids: List[Bid]):
        self.id = id
        self.title = title
        self.bids = bids

    def withdraw_bids(self, bids: List[Bid]):
        ...

    def make_a_bid(self, bid: Bid):
        ...

@property
def winners(self):
    ...
class DjangoAuctionsRepo(AuctionsRepo):
    def get(self, auction_id: int) -> Auction:
        auction_model = AuctionModel.objects.prefetch_related('bids').get(pk=auction_id)

        bids = [
            self._bid_from_model(bid_model)
            for bid_model in auction_model.bids.all()
315        return Auction(
            auction_model.id,
            auction_model.title,
            bids
        )
All that's left is to call `UseCase from Django` any framework
Clean Arch building blocks altogether
What to be careful of?

- non-idiomatic framework use
- more code (type hints help)
- copying data between objects
- validation?
- overengineering
When it pays off?

- lots of cases - testability
- delaying decision making - stay lean
- complicated domain
That's all, folks!

Questions?
Further reading

https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html


Clean Architecture Python (web) apps - Przemek Lewandowski

Software architecture chronicles - blog posts series

Boundaries - Gary Bernhardt

Exemplary project in PHP (blog post)

Exemplary project in PHP (repo)

Exemplary project in C# (repo)

Exemplary project in Python (repo)