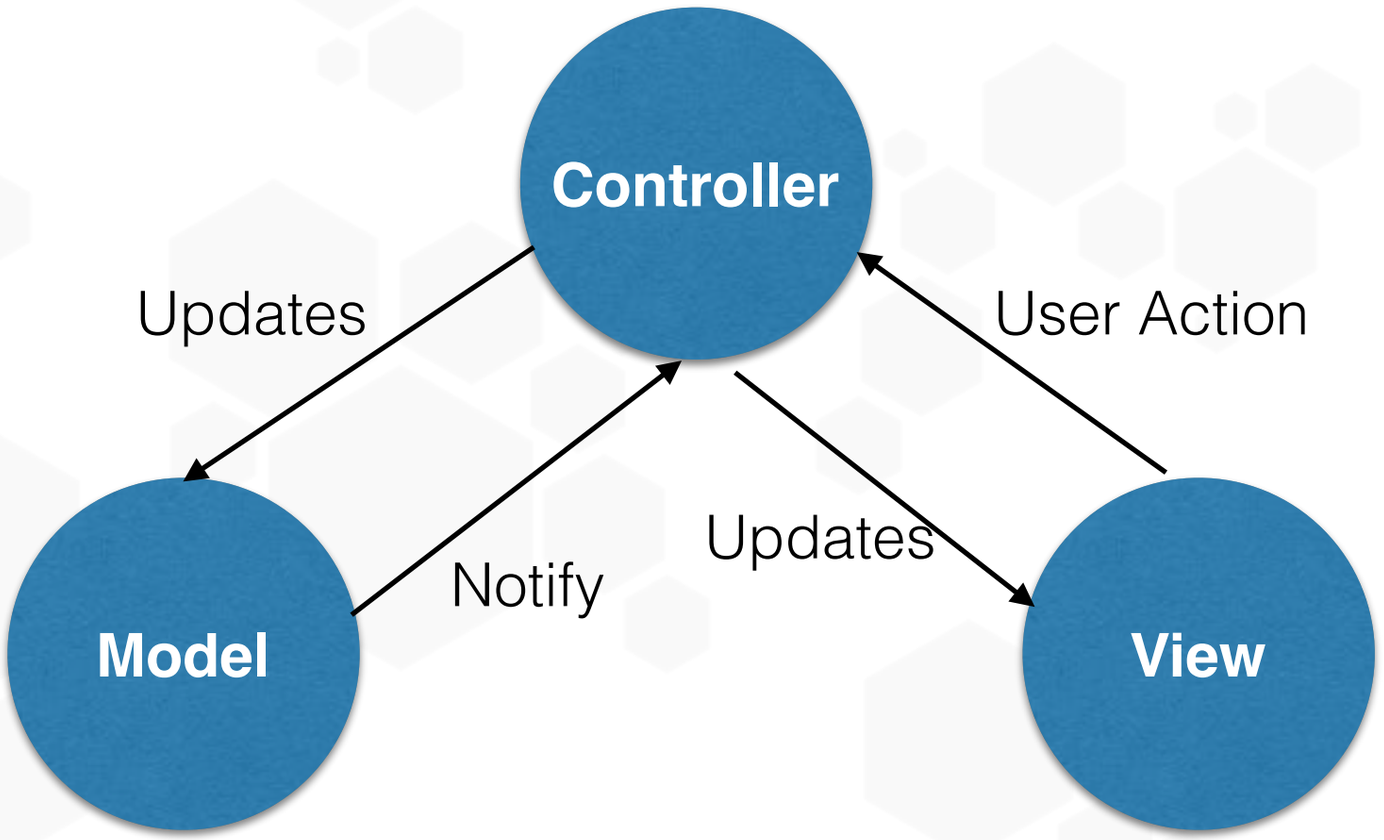
Hexagonal architectures in Django

Óscar Jesús García Pérez
@oscgrc - oscar@gagobox.com

Background: farielpuffs

MVC Pattern

Controller: Application logic

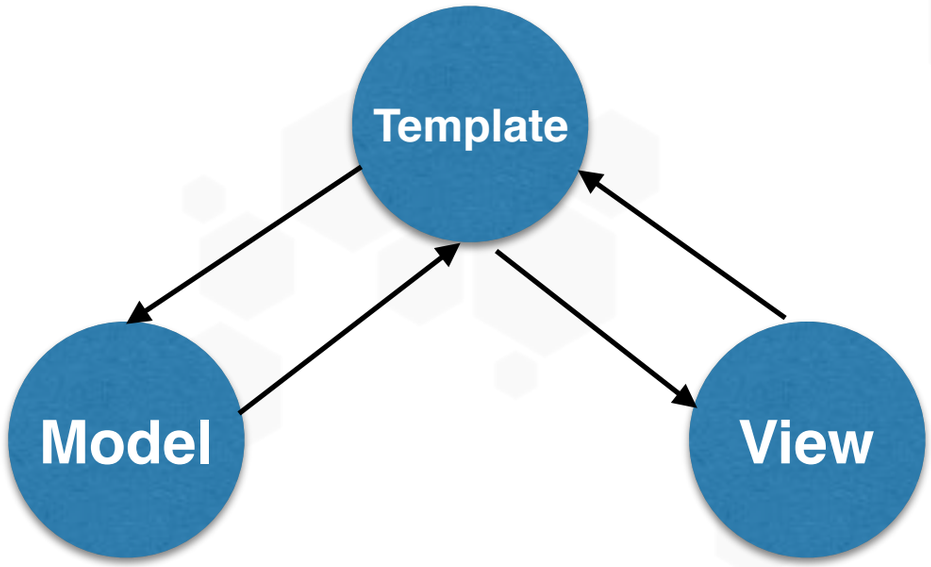


Model: Domain data an business rules

View: Representation of the information

Not in Django

- Model == Models
- Controller == Views
- View == Templates



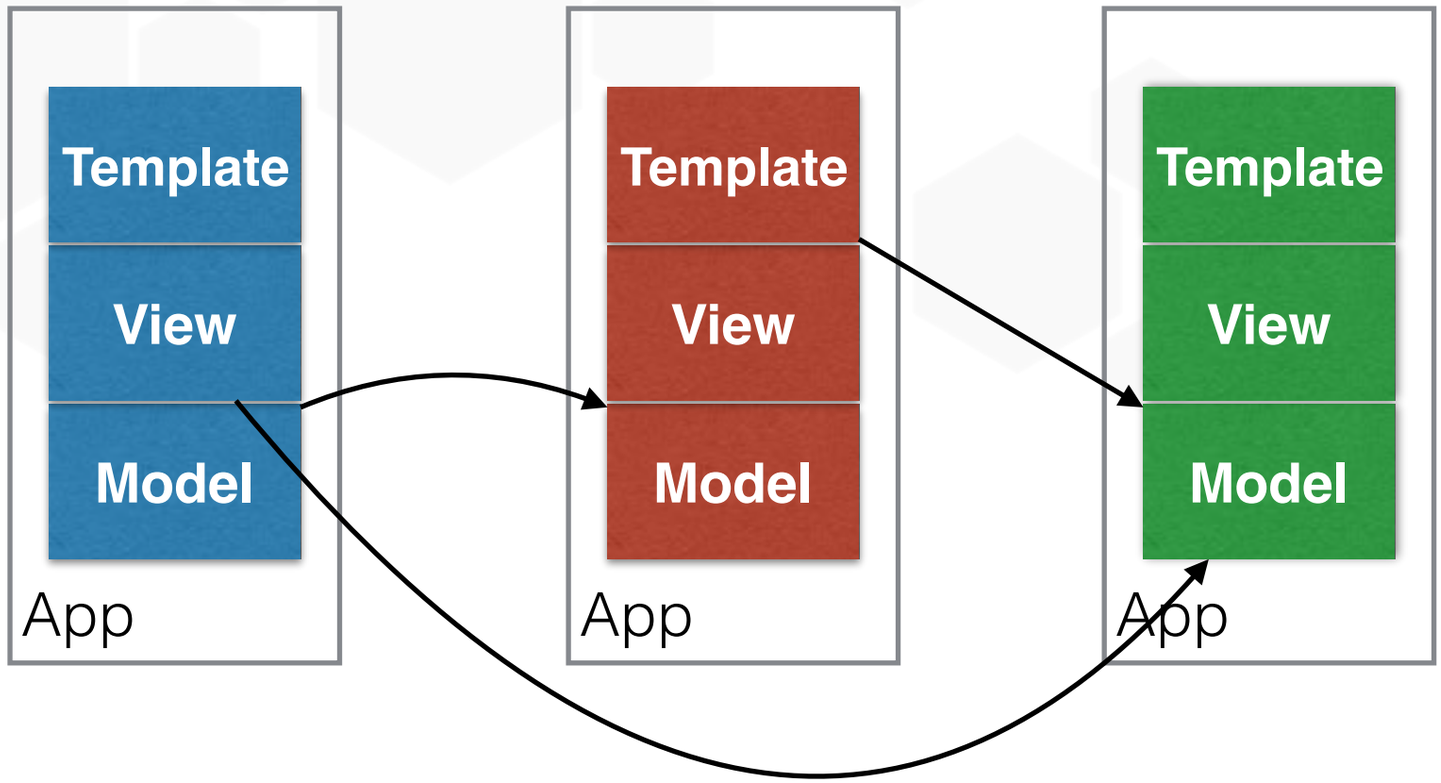
Django FAQ: MTV Framework

And there is more!

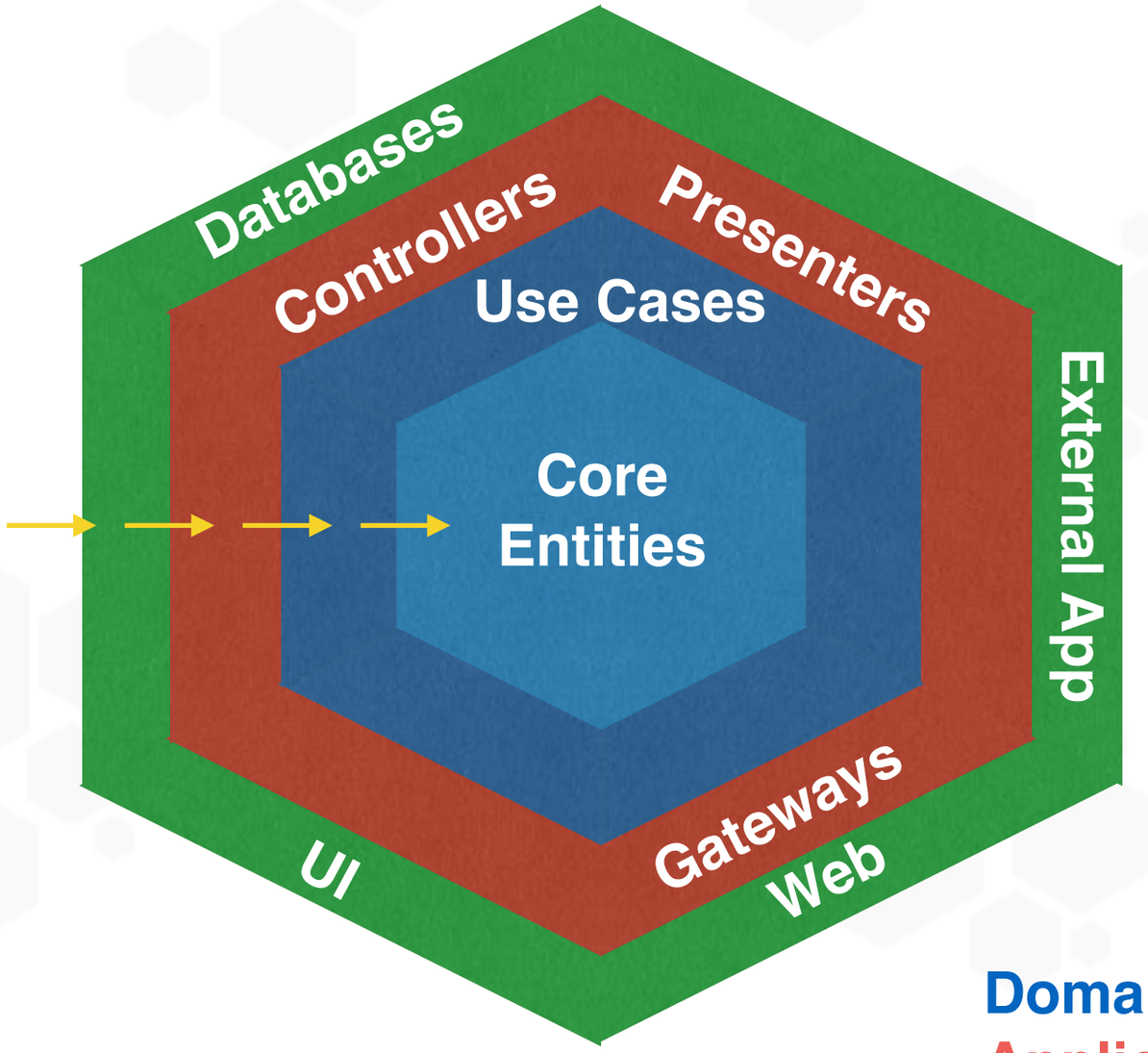
- Model
- Views
- Templates
- URLs
- Notifications/signals
- Forms
- Serializers
- Filters
- Permissions
- Tests

So, what is the problem?

- Queries inside the models.
- Models accessing models from other apps and creating circular dependencies.
- Losing control over where modifications on DB are being made.
- Difficult to test on isolation.
- Difficult to modify and add new features over time.



We can do better

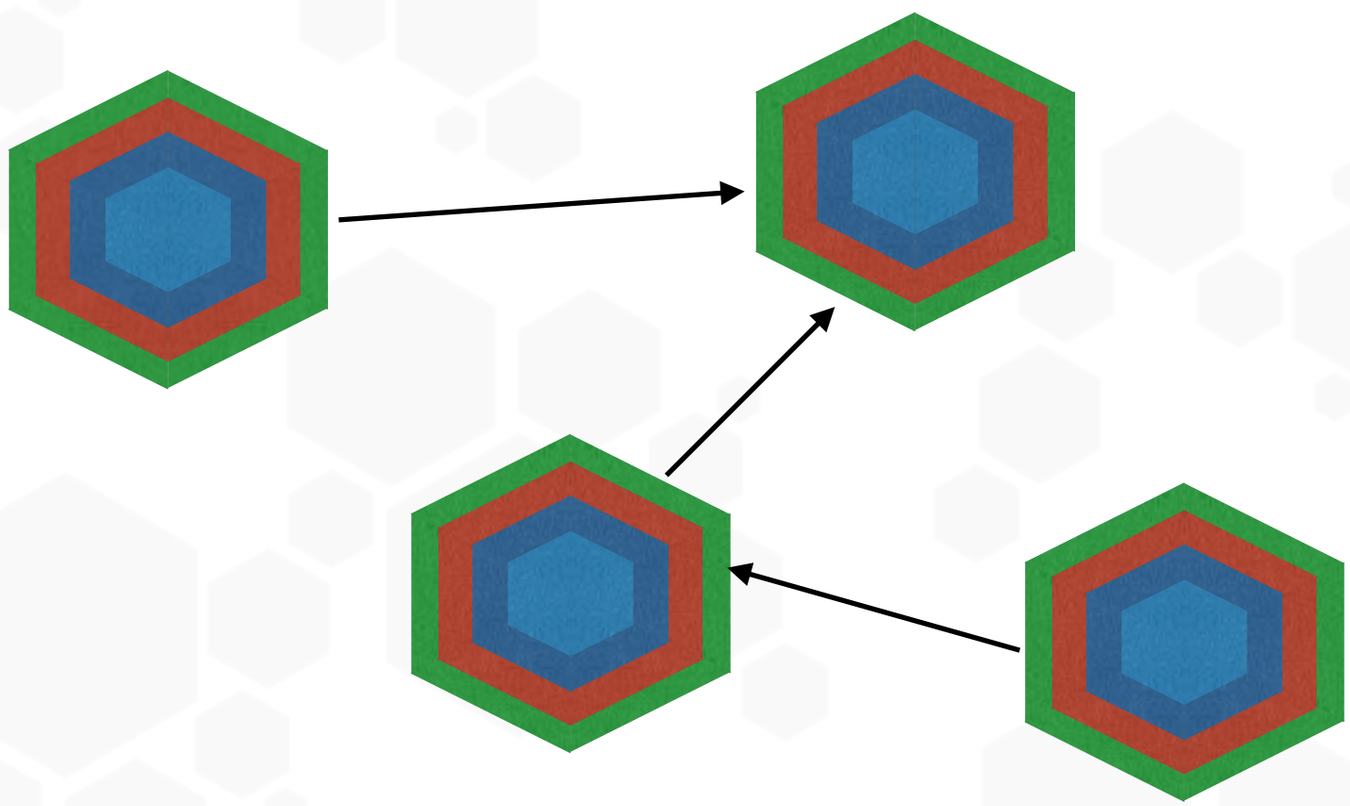


Domain level
Application level
Framework level (Django)

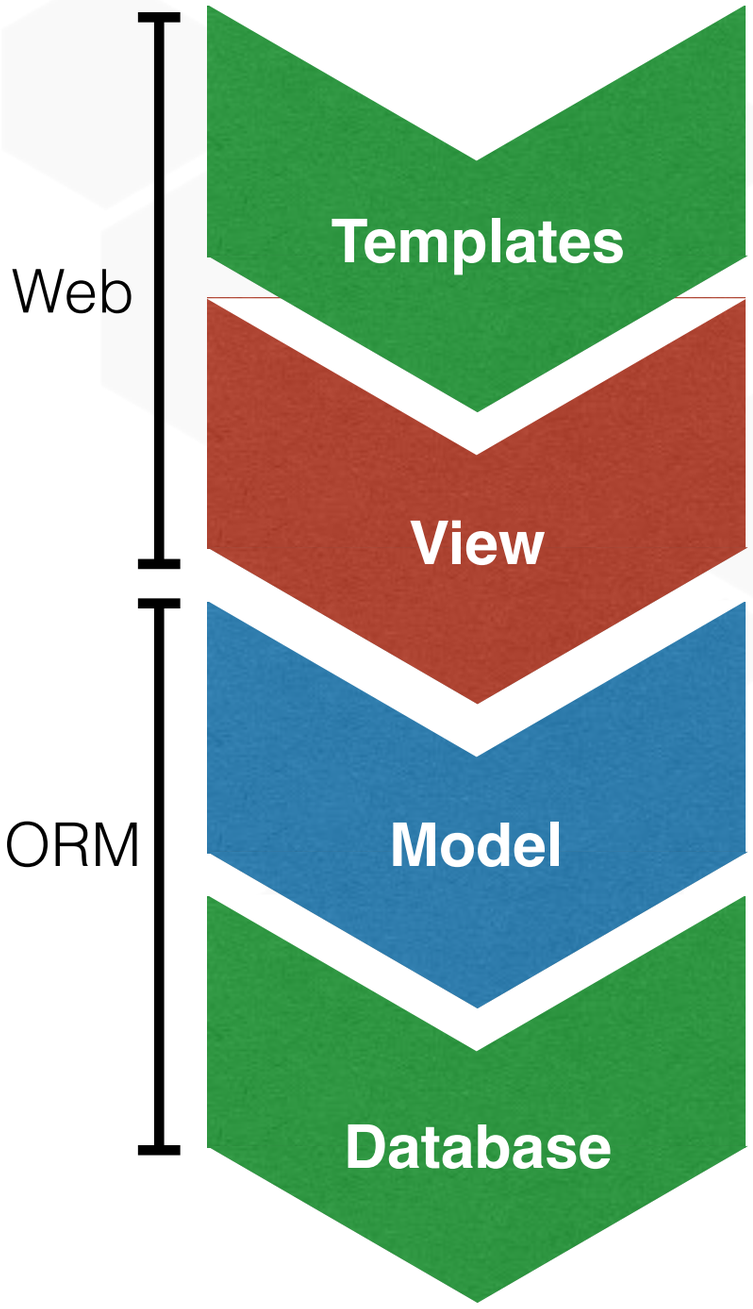
Ports and adapters

- Not limited to six faces only for representation purposes.
- Each layer access only its immediate inner layer.
- The inner layer provide “ports” to access its functionality (API).
- The outer layer create “adapters” to provide required data to the internal layer.

Hexagons talk to each other!



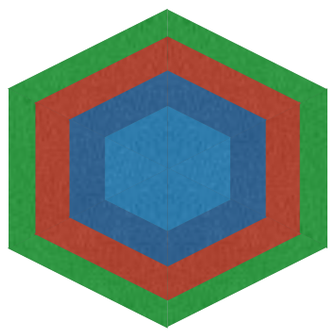
Common Django “problems”



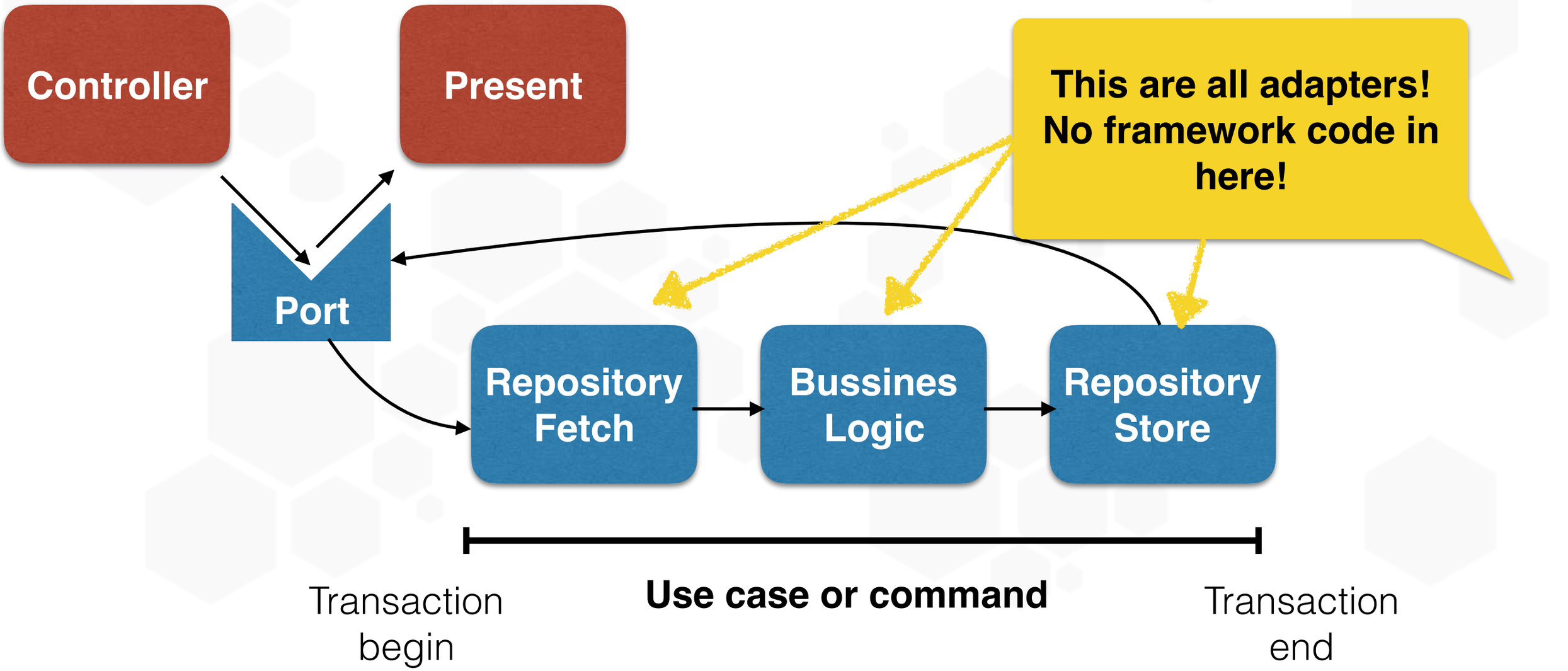
GUI
Business logic

Application logic
Presentation
Business logic
HTTP handling

Business logic
Managers, Querysets: Database Queries.



Domain-Driven Design



DDD building blocks

- Entities
- Value objects
- Aggregates
- Domain event
- Service
- Repository
- Factory

Show me the code!

https://github.com/Oscar-Garcia/django-hexarch-example

views.py

```python
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

```python
def vote(request, question_id):
    store = get_default_store()
    is_json = request.META.get('HTTP_ACCEPT', None) == 'application/json'
    view = PollsJSONView(request, store) if is_json else PollsHTMLView(request)
    return polls_controller.vote(store, view, question_id, request.POST.get('choice', -1))
```

domain/commands.py

```
 4 def vote(store, choice_id):
 5     choice = store.choices.get(choice_id)
 6     choice.votes = choice.votes + 1
 7     store.choices.save(choice)
```

framework/django_store.py

```
  1 # -*- coding: utf-8 -*-
  2 from django.core.exceptions import ObjectDoesNotExist
  3 from polls.models import Choice, Question
  4 from polls.domain.store import Store, StoreCollection
  5 from polls.domain.exceptions import NotFoundException
  6
  7
  8 class ModelAdapter(StoreCollection):
  9     def __init__(self, model_class):
 10         self._model_class = model_class
 11
>> 12     def get(self, id):
 13         try:
 14             return self._model_class.objects.get(pk=id)
 15         except ObjectDoesNotExist as error:
 16             raise NotFoundException(model=self._model_class, filters={'id': id}) from error
 17
 18     def save(self, entity):
 19         entity.save()
 20
 21
 22 class DjangoStore(Store):
 23
 24     @property
 25     def choices(self):
 26         return ModelAdapter(Choice)
 27
 28     @property
 29     def questions(self):
 30         return ModelAdapter(Question)
```

```
25
26 class MemoryAdapter(StoreCollection):
27     def __init__(self, model_class):
28         self._model_class = model_class
29         self._cache = {}
30
31     def all(self):
32         return self._cache.values()
33
>> 34     def get(self, id):
35         try:
36             return self._cache[int(id)]
37         except KeyError as error:
38             raise NotFoundException(model=self._model_class, filters={'id': id}) from error
39
40     def save(self, entity):
41         if not entity.id:
42             entity.id = len(self._cache) + 1
43         self._cache[entity.id] = entity
44
45     def save_batch(self, entities):
46         for entity in entities:
47             self.save(entity)
48
49
50 class QuestionMemoryAdapter(MemoryAdapter):
51
52     def __init__(self, model_class, store):
53         self.store = store
54         super().__init__(model_class)
55
56     def get_choices(self, question):
57         return [choice for choice in self.store.choices.all() if choice.question.id == question.id]
58
59
60 class MemoryStore(Store):
61
62     def __init__(self):
63         self._choices = MemoryAdapter(Choice)
64         self._questions = QuestionMemoryAdapter(Question, self)
65
66     @property
67     def choices(self):
68         return self._choices
69
70     @property
71     def questions(self):
72         return self._questions
```

Pragmatic approach

- Django models need to be adapted.
- Django Managers should be adapted as core factories and repositories.
- Handle the exceptions at the appropriate level. Django exceptions are at framework level.
- Entities should enforce its own consistency, but delay it till the last moment. Use guards.
- Forms validation better at application level.
- Easy to duplicate validation to reuse across different layers, than validate in one single place.
- Avoid overengineering.

Advantages

- Framework (Django) independence.
- Easy to test (isolation).
- Database independence.
- Run in headless mode.
- Good architecture helps to reduce technical debt.

Disadvantages

- Requires clear rules in the team and abilities to encapsulate and isolate.
- The design is more complex. Use a progressive approach.

Further reading

- Alistair Cockburn “Ports and Adapters”.
- Uncle Bob “Clean architectures”.
- Wikipedia Domain Driven Design.
- Django transactions.
- Poll example using Hexagonal Architecture.
- Overengineering Mistakes.
- What overengineering really is.