Don't start with a database

Practical clean architecture

Who I am

Grzegorz Kocjan

htd migawka.it

Goal of this talk

PRO TIP #0

New in Python

Typing

```
def get(order_id: int) -> dict:

    order_name: str = 'EuroPython2019!'

    order: dict = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Typing

```
def get(order_id: int) -> dict:

    order_name: str = 'EuroPython2019!'

    order: dict = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Typing

```
def get(order_id: int) -> dict:

    order_name: str = 'EuroPython2019!'

    order: dict = {
        'id': order_id,
        'name': order_name
    }

    return order
```

PRO TIP #1

Typing

```python
from typing import Dict, Union

Order = Dict[str, Union[int,str]]


def get(order_id: int) -> Order:

    order_name: str = 'EuroPython2019!'

    order: Order = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Typing

```
from typing import Dict, Union

Order = Dict[str, Union[int,str]]


def get(order_id: int) -> Order:

    order_name: str = 'EuroPython2019!'

    order: Order = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Typing

```
from typing import Dict, Union

Order = Dict[str, Union[int,str]]


def get(order_id: int) -> Order:

    order_name: str = 'EuroPython2019!'

    order: Order = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Typing

```python
from typing import Dict, Union

Order = Dict[str, Union[int,str]]


def get(order_id: int) -> Order:

    order_name: str = 'EuroPython2019!'

    order: Order = {
        'id': order_id,
        'name': order_name
    }

    return order
```

Python 3.7

Dataclasses

```
from dataclasses import dataclass



@dataclass
class Order:
    id: int
    name: str
```

Typing 💚 Dataclasses

```python
from our_package.entities import Order


def get(order_id: int) -> Order:

    order_name: str = 'EuroPython2019!'

    order: Order = Order(
        id=order_id,
        name=order_name,
    )

    return order
```

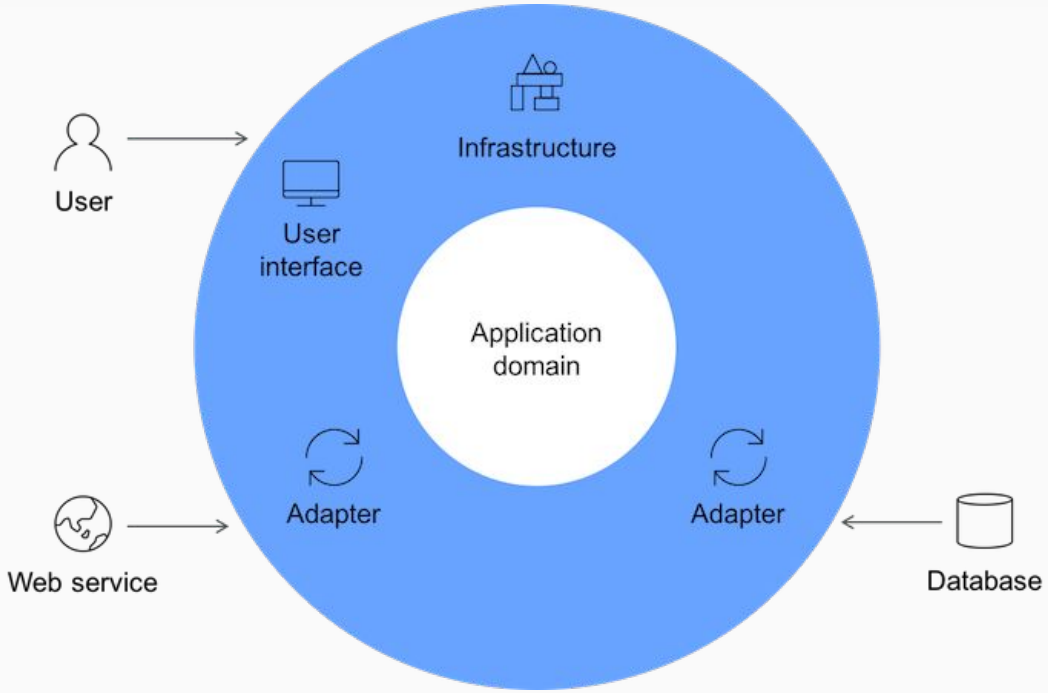
PRO TIP #2

Let's talk architecture

How to design architecture?

With the idea of maintaining it for the next 10 years

Clean architecture



Time for “coding”

Shop project - requirements

- Creating orders
- Viewing order lists
- Add existing items to order

Steps of building the application

- data definition - entities

Entities

```python
class BaseEntity:
    pass


@dataclass(frozen=True)
class Client(BaseEntity):
    id: int
    name: str
```

Entities

```
class BaseEntity:
    pass


@dataclass(frozen=True)
class Client(BaseEntity):
    id: int
    name: str
```

Entities - frozen=True

```
>>> client = Client(id=1, name='Grzegorz')

>>> client.name = 'Alice'


Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<string>", line 3, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

Entities

```python
@dataclass(frozen=True)
class Product(BaseEntity):
    id: int
    name: str
    price: float
```

Entities

```
@dataclass(frozen=True)
class Order(BaseEntity):
    id: int
    created: datetime
    client: Client
    total_cost: float = 0.0
    items: ??
```

Entities

```
@dataclass(frozen=True)
class Order(BaseEntity):
    id: int
    created: datetime
    client: Client
    total_cost: float = 0.0
    items: List[Product] = field(default_factory=list)
```

Steps of building the application

- data definition - entities

Steps of building the application

- data definition - entities
- use cases

Use cases

```
class OrderLogic:
    def create(self, client_id: int) -> Order:
        ...

    def search(self, client_id: int) -> List[Order]:
        ...

    def add_product(self, order_id: int, product_id: int) ->
Order:
        ...
```

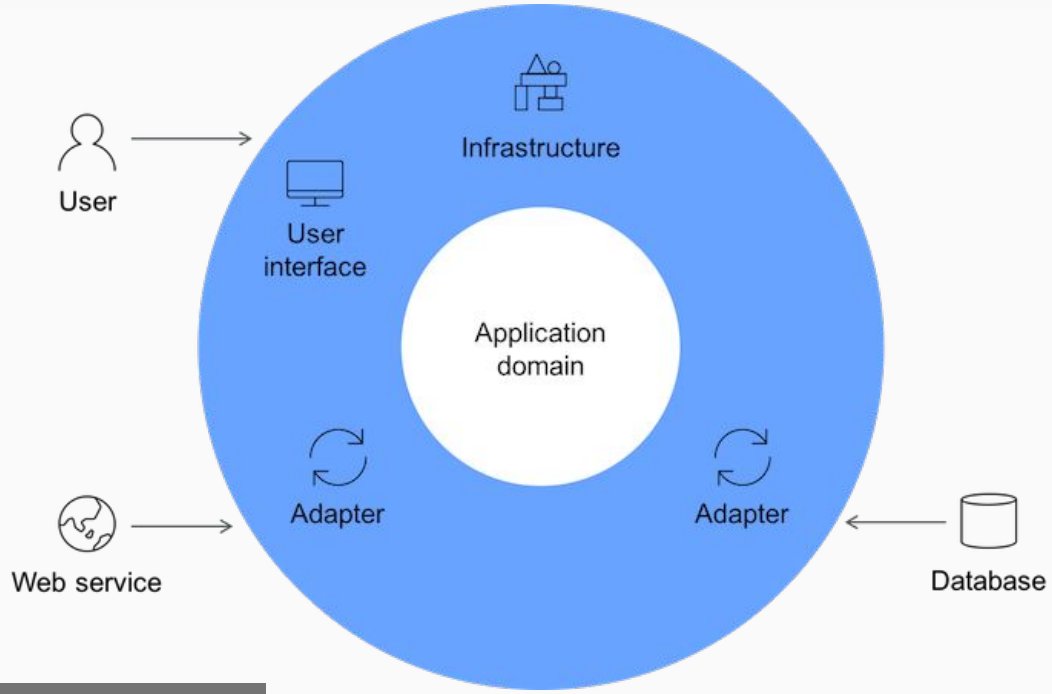
But those are operations on a database …

… it's time to introduce …

An abstraction

PRO TIP #3

Clean architecture



Repositories

```python
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```python
class IClientRepository(abc.ABC):

    @abc.abstractmethod
    def create(self, name: str) -> Client:
        pass

    @abc.abstractmethod
    def get(self, client_id: int) -> Client:
        pass
```

Repositories

```python
class IProductRepository(abc.ABC):

    @abc.abstractmethod
    def get(self, product_id: int) -> Product:
        pass
```

Repositories

```python
class IOrderRepository(abc.ABC):

    def create(self, client: Client) -> Order:

    def get(self, order_id: int) -> Order:

    def save(self, order: Order) -> Order:

    def search(
        self, client: Optional[Client] = None
    ) -> List[Order]:
```

Use cases

```
class OrderLogic:
    def create(self, client_id: int) -> Order:
        ...

    def search(self, client_id: int) -> List[Order]:
        ...

    def add_product(self, order_id: int, product_id: int) ->
Order:
        ...
```

Use cases

```python
class OrderLogic:

    def __init__(
        self,
        orders: IOrderRepository,
        products: IProductRepository,
        clients: IClientRepository,
    ) -> None:
        self._orders: IOrderRepository = orders
        self._products: IProductRepository = products
        self._clients: IClientRepository = clients
```

Use cases

```
class OrderLogic:
    @inject
    def __init__(
        self,
        orders: IOrderRepository,
        products: IProductRepository,
        clients: IClientRepository,
    ) -> None:
        self._orders: IOrderRepository = orders
        self._products: IProductRepository = products
        self._clients: IClientRepository = clients

from injector import inject
```

PRO TIP #4

Use cases

```
class OrderLogic:

    def search(self, client_id: int) -> List[Order]:
        client = self._clients.get(client_id)
        return self._orders.search(client=client)

    def create(self, client_id: int) -> Order:
        client = self._clients.get(client_id)
        return self._orders.create(client=client)
```

Use cases

```
class OrderLogic:

    def add_product(self, order_id: int, product_id: int) ->
Order:
        order = self._orders.get(order_id)
        product = self._products.get(product_id)
```

Use cases

```
class OrderLogic:

    def add_product(self, order_id: int, product_id: int) ->
Order:
        order = self._orders.get(order_id)
        product = self._products.get(product_id)

        order = replace(  # from dataclasses import replace
            order,
            items=order.items + [product],
            total_cost=order.total_cost + product.price,
        )
```

Use cases

```
class OrderLogic:

    def add_product(self, order_id: int, product_id: int) ->
Order:
        order = self._orders.get(order_id)
        product = self._products.get(product_id)

        order = replace(  # from dataclasses import replace
            order,
            items=order.items + [product],
            total_cost=order.total_cost + product.price,
        )

        return self._orders.save(order)
```

So much work…

why bother?

Tests

```
def test_add_product_increases_order_total_cost(

) -> None:
```

PRO TIP #5

Tests

```
def test_add_product_increases_order_total_cost(
    prepare_repositories: StaticRepositories
) -> None:




StaticRepositories = Tuple[
    IOrderRepository, IProductRepository, IClientRepository
]
```

PRO TIP #6

Tests

```
def test_add_product_increases_order_total_cost(
    prepare_repositories: StaticRepositories
) -> None:
```

Tests

```
def test_add_product_increases_order_total_cost(
    prepare_repositories: StaticRepositories
) -> None:

1:     logic = OrderLogic(*prepare_repositories)

2:     order = logic.add_product(order_id=1, product_id=1)

3:     assert order.total_cost == 100
```

Steps of building the application

- data definition - entities
- use cases

Steps of building the application

- data definition - entities
- use cases
- implement interfaces

Time for a database?

No no no :)

A repository

A repository

A repository in

A repository in

A repository in memory

Repozytorium w pamięci

```
class RamStorage(Generic[T]):
    def __init__(self) -> None:
        self._storage: StorageType = {}
```

```
T = TypeVar("T")
StorageType = Dict[int, T]
```

```
RamStorage[Client]()
RamStorage[Order]()
```

Repozytorium w pamięci

```
class RamStorage(Generic[T]):
    def __init__(self) -> None:
        self._storage: StorageType = {}

    def add(self, item: T) -> None:

    def get(self, pk: int) -> Optional[T]:

    def search(self, **kwargs: Any) -> RamStorage[T]:

    def remove(self, item: T) -> None:

    def all(self) -> List[T]:
```

Repozytorium w pamięci - wykorzystanie

```
class ProductRepository(IProductRepository):
    def __init__(self) -> None:
        self._ram_storage = RamStorage[Product]()
```

Repozytorium w pamięci - wykorzystanie

```
class ProductRepository(IProductRepository):
    def __init__(self) -> None:
        self._ram_storage = RamStorage[Product]()

    def get(self, product_id: int) -> Product:
        result = self._ram_storage.get(product_id)
        if result is None:
            raise ProductNotFound()
        return result
```

Steps of building the application

- data definition - entities
- use cases
- implement interfaces

Steps of building the application

- data definition - entities
- use cases
- implement interfaces
- API

Choose framework!

and this is slide no 77!

PRO TIP #7

Flask + Connexion

Flask + connexion - api.yaml

```yaml
paths:
  /orders/search/:
    get:
      operationId: our_package.endpoints.search
      parameters:
          ...
      responses:
          ...
```

Flask + connexion - api.yaml

The shop

orders

Show/Hide | List Operations | Expand Operations

POST	/orders/create/
GET	/orders/search/
PUT	/orders/{order_id}/add_product/{product_id}

[BASE URL: /api/ , API VERSION: 1.0]

Flask + connexion - endpoints

```
def search(logic: OrderLogic, client_id: int) -> List[Order]:
    return logic.search(client_id=client_id)

def create(logic: OrderLogic, body: dict) -> Order:
    return logic.create(client_id=body['client_id'])

def add_product(logic: OrderLogic, order_id: int, product_id:
int) -> Order:
    return logic.add_product(order_id, product_id)
```

Flask + connexion - endpoints

```
def search(logic: OrderLogic, client_id: int) -> List[Order]:
    return logic.search(client_id=client_id)

def create(logic: OrderLogic, body: dict) -> Order:
    return logic.create(client_id=body['client_id'])

def add_product(logic: OrderLogic, order_id: int, product_id:
int) -> Order:
    return logic.add_product(order_id, product_id)
```

Injector

Flask + connexion - endpoints

```
flask_injector = FlaskInjector(
    app=app,
    modules=[MyModule]
)
app.config["FLASK_INJECTOR"] = flask_injector
```

Flask + connexion - endpoints

```
class MyModule(injector.Module):
    def configure(self, binder: injector.Binder) -> None:
```

Flask + connexion - endpoints

```
class MyModule(injector.Module):
    def configure(self, binder: injector.Binder) -> None:
        binder.bind(OrderLogic, to=OrderLogic)
```

Flask + connexion - endpoints

```
class MyModule(injector.Module):
    def configure(self, binder: injector.Binder) -> None:
        binder.bind(OrderLogic, to=OrderLogic)
        binder.bind(
            IClientRepository,
            to=ClientRepository,
            scope=injector.SingletonScope
        )
```

Flask + connexion - endpoints

```
def search(logic: OrderLogic, client_id: int) -> List[Order]:
    return logic.search(client_id=client_id)

def create(logic: OrderLogic, body: dict) -> Order:
    return logic.create(client_id=body['client_id'])

def add_product(logic: OrderLogic, order_id: int, product_id:
int) -> Order:
    return logic.add_product(order_id, product_id)
```

Flask + connexion - endpoints

```
def search(logic: OrderLogic, client_id: int) -> List[Order]:
    return logic.search(client_id=client_id)

def create(logic: OrderLogic, body: dict) -> Order:
    return logic.create(client_id=body['client_id'])

def add_product(logic: OrderLogic, order_id: int, product_id:
int) -> Order:
    return logic.add_product(order_id, product_id)
```

Response serialization

Flask + connexion - encoder

```
class ApiJsonEncoder(JSONEncoder):
```

Flask + connexion - encoder

```
class ApiJsonEncoder(JSONEncoder):
    def default(self, obj: Any) -> Any:
        if isinstance(obj, (datetime.datetime, datetime.date)):
            return obj.isoformat()
```

Flask + connexion - encoder

```
class ApiJsonEncoder(JSONEncoder):
    def default(self, obj: Any) -> Any:
        if isinstance(obj, (datetime.datetime, datetime.date)):
            return obj.isoformat()

        if isinstance(obj, Client):
            return {"id": obj.id, "name": obj.name}
```

Flask + connexion - encoder

```
class ApiJsonEncoder(JSONEncoder):
    def default(self, obj: Any) -> Any:
        if isinstance(obj, (datetime.datetime, datetime.date)):
            return obj.isoformat()

        if isinstance(obj, BaseEntity):
            return {key: value for key, value in vars(obj).items()
if value is not None}
```

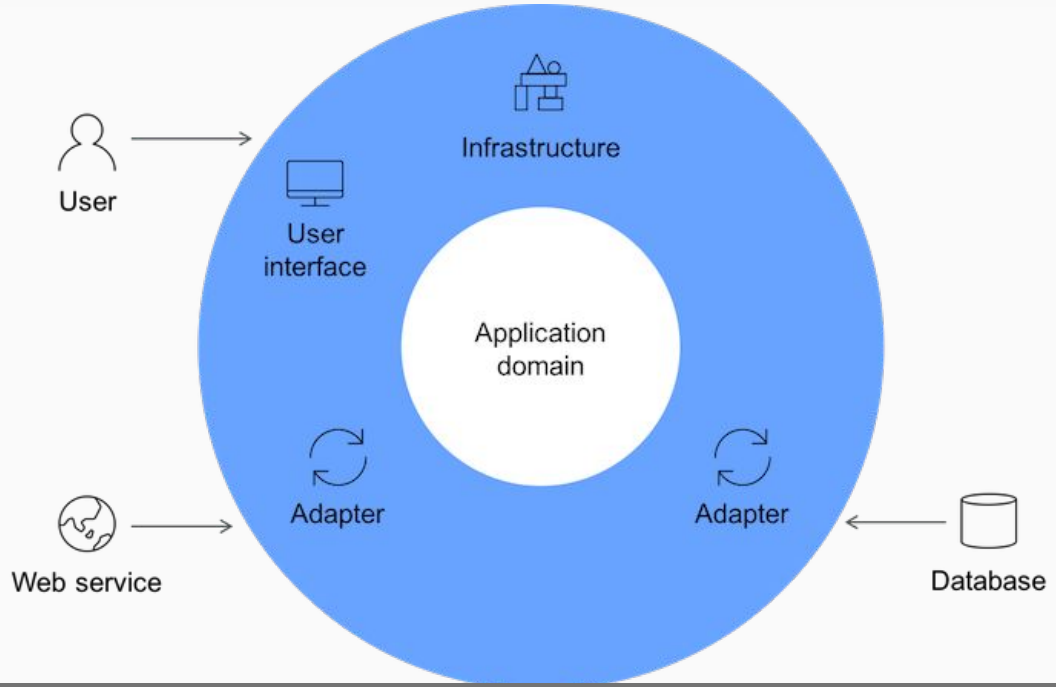
Flask + connexion - encoder

```
class ApiJsonEncoder(JSONEncoder):
    def default(self, obj: Any) -> Any:
        if isinstance(obj, (datetime.datetime, datetime.date)):
            return obj.isoformat()

        if isinstance(obj, Base):
            return {key: value for key, value in vars(obj).items()
if value is not None}

        return JSONEncoder.default(self, obj)
```

Steps of building the application

- data definition - entities
- use cases
- implement interfaces
- API

Clean architecture



Protect borders

Borders - project structure

- `shop`
- `ram_db`
- `api`
- `setup.py`

Borders - project structure

- shop
  - setup.py
  - requirements.txt
- ram_db
  - setup.py
  - requirements.txt
- api
  - setup.py
  - requirements.txt
- setup.py

Granice - struktura projektu

- shop
  - setup.py
  - requirements.txt -> injector
- ram_db
  - setup.py
  - requirements.txt -> shop
- api
  - setup.py
  - requirements.txt -> shop, ram_db, flask, connexion
- setup.py

Borders - project structure

- shop
  - setup.py
  - requirements.txt
- ram_db
  - setup.py
  - requirements.txt
- api
  - setup.py
  - requirements.txt
- setup.py

DEMO

Benefits

- Business logic independence
- Ease of technology update/change
- Clear and secure borders
- Technology chosen based on knowledge
- Fast prototyping / POC
- Ease of testing
- Installing only required packages

Architecture is a set of conscious decisions

Clean architecture is a way to delay important decisions

Further reading

Clean Architecture - Robert C. Martin

https://g.co/kgs/kbaamc

Python microlibs:

https://medium.com/@jherreras/python-microlibs-5be9461ad979

Thank you

grzegorz@kocjan.me
@GrzegorzKocjan

https://migawka.it/europython2019

htd migawka.it