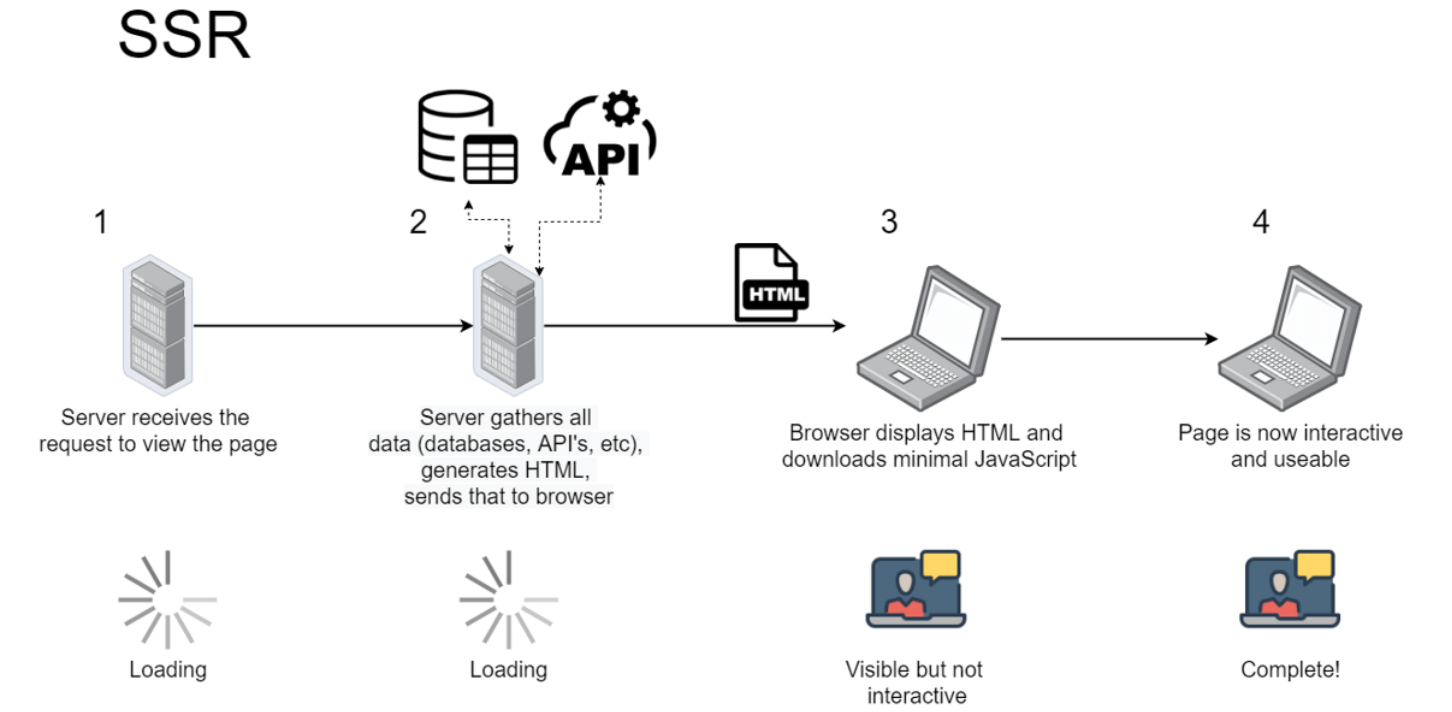
# Arhitektura veb aplikacija

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

# Tipovi arhitektura veb aplikacija

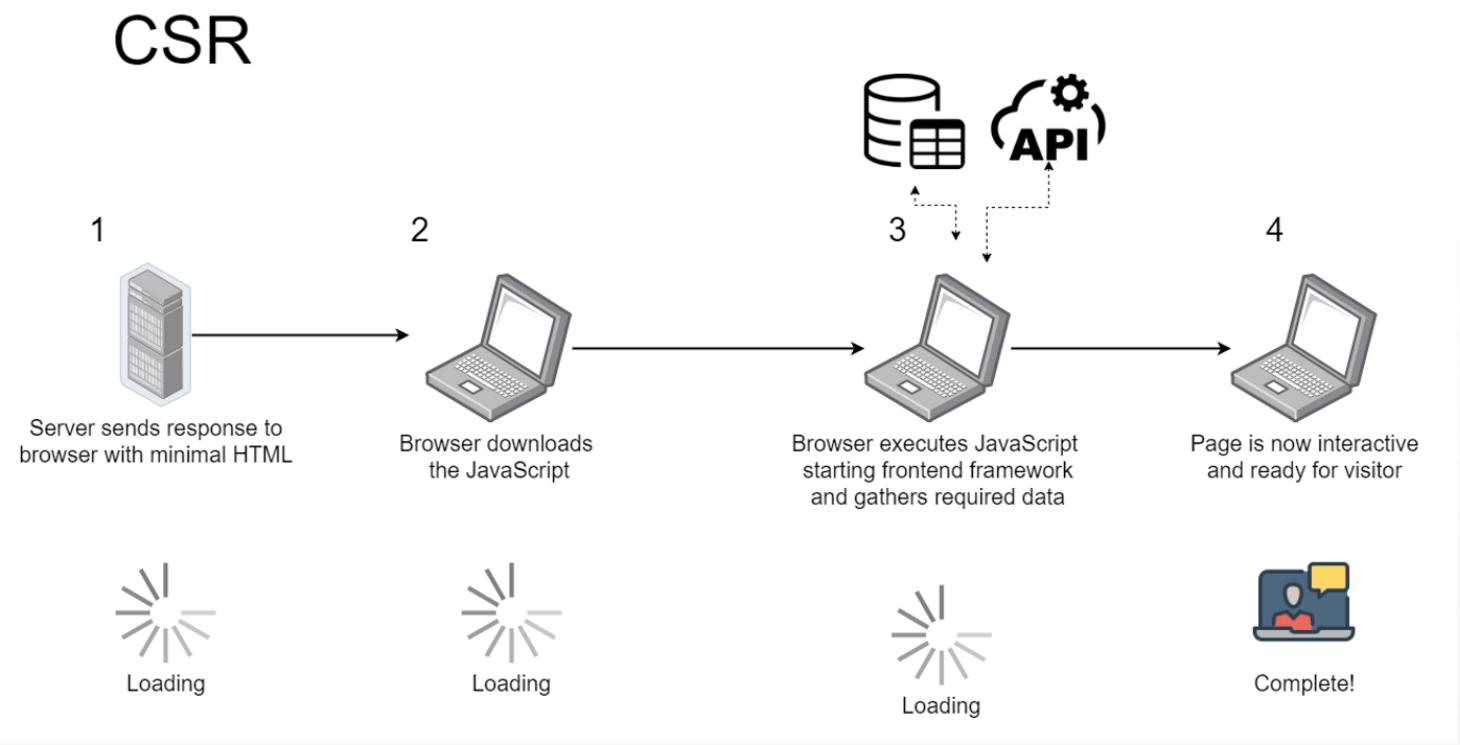
# Moderna Veb Arhitektura – Gde se izvršava logika?

- **Osnovna dilema**: Ko generiše HTML? Server ili Klijent (Browser)?
- **Server-Side Rendering** (**SSR**): Klasičan pristup (npr. Django MVT). Server šalje gotovu stranicu.
  - *Prednost*: Brz inicijalni prikaz, SEO optimizovano, jednostavnija sigurnost podataka.



# Moderna Web Arhitektura – Gde se izvršava logika?

- **Client-Side Rendering** (**CSR**): Moderni pristup (npr. React, Vue). Server (npr. Django REST) šalje JavaScript, klijent gradi stranicu. Kasnije server šalje samo podatke (json).
  - Prednost: Fluidno korisničko iskustvo (bez osvežavanja stranice), smanjeno opterećenje



# Moderna Web Arhitektura – Gde se izvršava logika?

- CSR vs SSR

| Karakteristika | SSR (Django) | CSR (React/SPA) |
|---|---|---|
| Inicijalno učitavanje | Brže (Server šalje gotov HTML) | Sporije (Mora se učitati sav JS) |
| Interaktivnost | Sporija (Svaki klik je novi Request) | Trenutna (Menja se samo deo DOM-a) |
| Kompleksnost | Manja (Sve je u jednom projektu) | Veća (Odvojeni Backend API i Frontend) |
| AI Integracija | Lakša za statičke analize | Bolja za interaktivne AI agente (Chat) |

- **“Rule of thumb”:** Ako pravite informativni portal ili e-commerce -> SSR. Ako pravite kompleksnu alatku (npr. Google Sheets, Dashboard) -> CSR

# Moderna Web Arhitektura – Gde se izvršava logika?

- **Hibridni modeli**: Današnji standard (npr. Next.js) koji kombinuje najbolje od oba sveta.
  - Umesto da birate jednu strategiju za ceo sajt, možete birati za svaku stranicu/kompnentu kako će se procesirati.
- Proces hibridnog renderovanja:

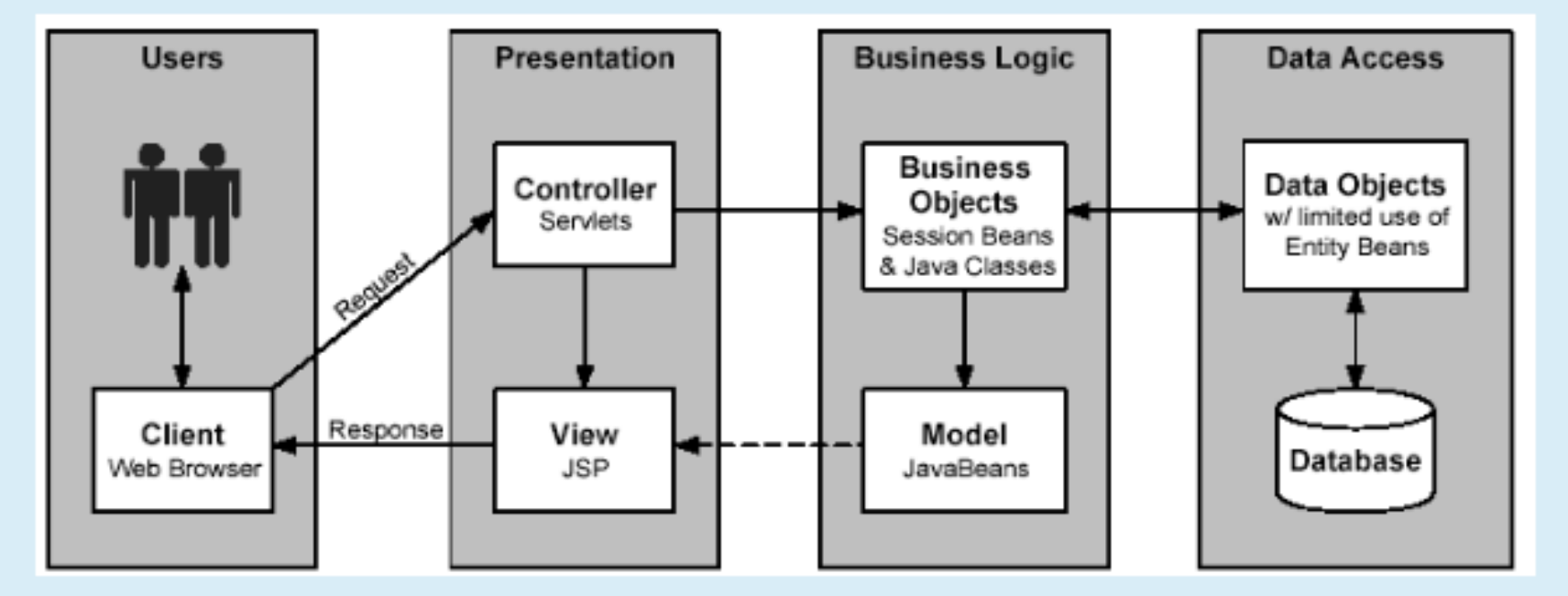
1. Inicijalni prikaz – server pre-renderuje React komponente i šalje statični html klijentu koji ga odmah prikazuje (dobre metrike SEO i LCP).
2. Hidratacija – pregledač u pozadini preuzima javascript i "oživljava" statični HTML tako što na njega kači event listenere i postavlja stanje aplikacije bez ponovnog iscrtavanja celog DOM-a.
3. Interaktivnost I navigacija – nakon hidratacije, aplikacija se ponaša kao single page aplikacija: akcije klika na linkove, popunjavanja formi, filtriranja podataka itd obavljaju se isključivo na klijentu (nema reloada cele stranice). I dalje neki delovi sajta na primer javni, katalog proizvoda i slično mogu biti SSR radi pretraživača.

# CSR arhitektura i AI

- **Struktura podataka**: CSR arhitektura primorava inženjera da pravi čiste JSON API-je.
- **AI kao klijent**: Čist API omogućava da vašu aplikaciju ne koriste samo ljudi kroz browser, već i AI agenti koji direktno čitaju podatke.
- **Generisanje interfejsa**: AI modeli mnogo lakše generišu React komponente (CSR) nego što razumeju kompleksne serverske template-ove pomešane sa biznis logikom.
- **Zaključak**: Razumevanje gde se "vrti" kod omogućava vam da pravilno pozicionirate AI module (npr. da li AI sumira tekst na serveru pre slanja ili React poziva AI asinhrono nakon učitavanja).

# Višeslojna Arhitektura – Više od podele koda

- Osnovni koncept: Razbijanje aplikacije na logičke celine (Slojeve) koje komuniciraju samo sa susednim slojevima.
- Standardni slojevi:
  1. Presentation (UI): Django Templates ili React – Interakcija sa korisnikom.
  2. Service/Business Logic: Srce aplikacije (Django Views, Django Rest) – Ovde se donose odluke.
  3. Data Access (Persistence): Django ORM – Rad sa bazom podataka.



# Višeslojna Arhitektura – Više od podele koda

- Nekada se **slojevima** (**layers**) opisuje logička podela aplikacije (način organizacije koda), dok se **nivoima** (**tiers**) opisuje organizacija gde se pojedine komponente nalaze na različitim fizičkim serverima (gde se kod izvršava).
- Na primer, aplikacija kojoj je GUI posebna (npr. React) aplikacija koja komunica sa backend (npr. Django) serverom koji komunicira sa posebnim database serverom je **tronivovska** aplikacija.
- To i dalje ne treba mešati sa podelom na **monolitne** i **distribuirane** aplikacije. Goreopisana aplikacija je monolitna ako ima jednu bazu koda (npr. na githubu) i jedinstveni proces pravljenja (jedan tim).

# Gde se uklapaju AI servisi u logičku podelu po slojevima?

- **Problem**: AI modeli su "nepouzdani" (halucinacije, latencija).
  - AI logika nikada ne sme biti direktno u UI sloju niti u bazi. Ona pripada isključivo Servisnom sloju ili posebnom AI sloju.
- **Rešenje**: Validation & Sanitization Layer.
  - Svaki podatak koji stigne iz AI-a (npr. sažetak posta) mora proći kroz servisnu proveru pre nego što ode u bazu ili ka korisniku.
- **Inženjerski zadatak**: Projektovati slojeve tako da aplikacija može da radi (u "safe mode" režimu) čak i ako AI servis privremeno nije dostupan.
- **Razdvajanje nadležnosti** (Separation of Concerns):
  - Pitanje: Da li View treba da zna da koristimo GPT-4?
  - Odgovor: Ne. View samo traži get_summary(), a Servisni sloj odlučuje kako će to nabaviti.

# Višeslojna arhitektura kao "Ograda" za AI

- **Koncept izolovanog okruženja** (Sandboxing):
  - Višeslojna arhitektura arhitektura služi kao ograda koja sprečava da greška u AI generisanom modulu sruši celu bazu podataka ili kompromituje sigurnost.
- **Zaštita integriteta**:
  - Sloj baze (Data Access) mora imati sopstvene validacije (Constraints) koje AI ne može da zaobiđe.
- Arhitektura je jedini način da **zadržite kontrolu** nad sistemom u kojem značajan deo koda može biti generisan automatizovano.

# Distribuirane veb aplikacije

- Distribuirana arhitektura: Šira kategorija (uključujući mikroservise) gde su delovi sistema fizički odvojeni i komuniciraju putem mrežnih protokola da bi radili zajedno.

| Feature | Monolithic Application | Distributed Application |
|---|---|---|
| **Deployment** | Deployed as a single unit (one EXE, JAR, or WAR file). | Composed of multiple independent units (services) deployed separately. |
| **Communication** | Functions call each other in-memory; near-instantaneous. | Components communicate over a network (APIs, gRPC, Message Queues). |
| **Scaling** | You must scale the entire app at once, even if only one part is slow. | You can scale specific services independently based on demand. |
| **Failures** | A single bug or crash can take down the entire system. | Failure in one service is often isolated; other services remain active. |
| **Data** | Typically uses a single shared database for all functions. | Often uses distributed databases or unique databases for each service. |

# Još dve (pod)kategorije veb arhitektura

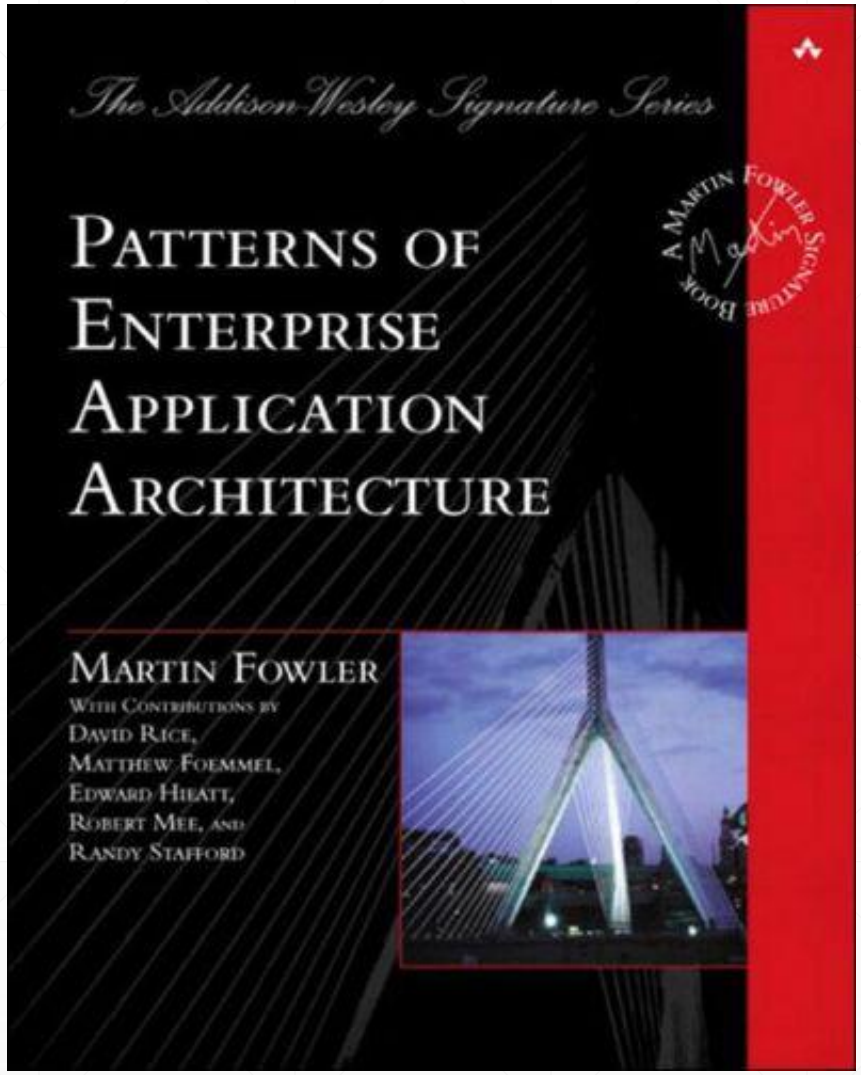
- **PWA (Progresivne veb aplikacije):** Koriste moderne web API-jeve i Service Workere kako bi omogućile rad van mreže (offline) i instalaciju na uređaj, gradeći se na osnovama CSR-a, predstavlja nadogradnju korisničkog iskustva (UX).
- **Serverless Web Apps:** Arhitekturni pristup gde se backend logika izvršava u "funkcijama" (FaaS) koje pokreće događaj, čime se infrastruktura apstrahuje, a aplikacija se skalira automatski.

# Obrasci za arhitekturu veb aplikacija

# Projektni obrasci/uzorci (Design patterns)

- *"Svaki uzorak opisuje **problem** koji se javlja iznova i iznova u našem okruženju, a zatim opisuje **suštinu rešenja** tog problema, na takav način da se takvo rešenje može **koristiti ponovo** milion puta, a da se nikada to ne uradi dva puta na isti način“*
  - Christopher Alexander, poznati građevinski arhitekta

# Izvori informacija o projektnim uzorcima



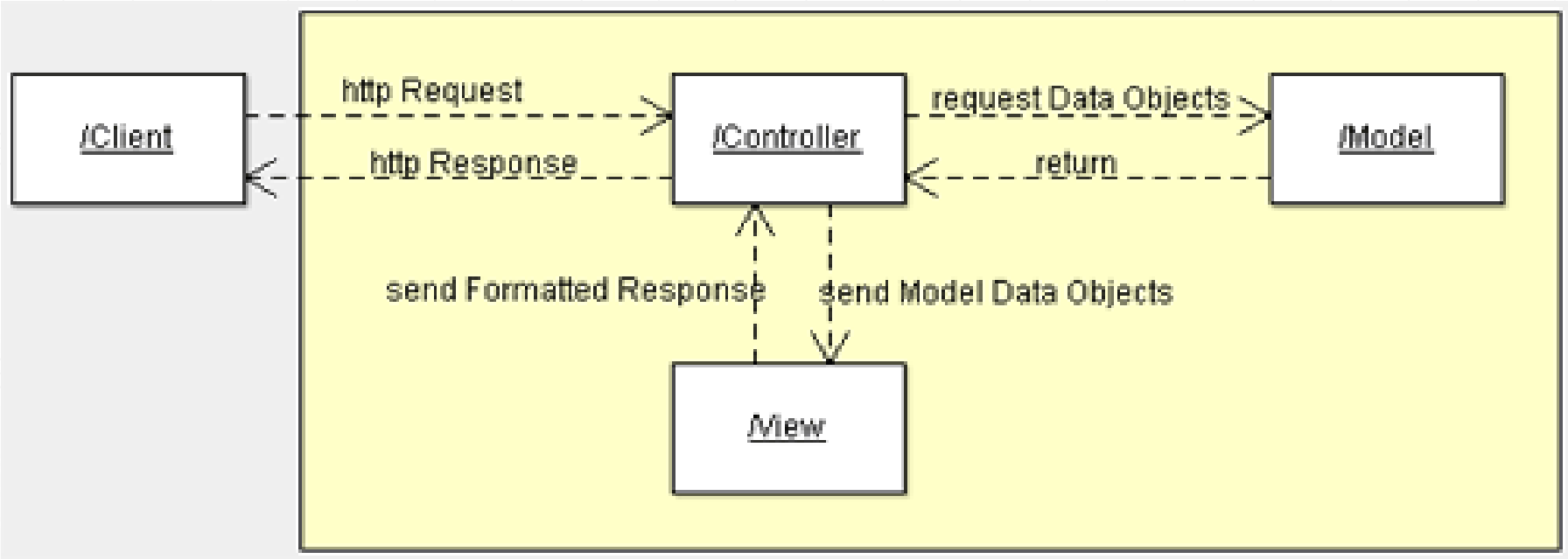

- PoEAA patterns
  - Razmotrićemo samo neke uzorke iz ove knjige, koji se često koriste u web programiranju
  - Za razliku od GoF uzoraka, koji su uglavnom na nivou detaljnog projektovanja, PoEAA se odnosi na nivo arhitekture
- Ažurnija verzija Fowlerovih obrazaca je na sajtu:

  https://martinfowler.com/architecture/

# Obrazac Model View Controller

- **MVC uzorak** razdvaja aplikaciju u 3 celine: Model, Prikaz i Kontroler:
- **Model** je odgovoran za upravljanje podacima, on čuva i vraća entitete korišćene od strane aplikacije, obično iz baze podataka, i sadrži logiku aplikacije.
- P**rikaz** (prezentacija) je odgovoran za prikazivanje podataka koje pruža model u određenom formatu. On ima sličnu upotrebu kao šabloni prisutni u nekim popularnim web aplikacijama, kao što su WordPress, Joomla, ...
- **Kontroler** upravlja modelom i prikazom da rade zajedno. Kontroler primi zahtev od klijenta, pokreće model za obavljanje traženih poslova i šalje podatke na Prikaz. Prikaz formatira podatke za prikaz korisniku, u HTML formatu.

# Dijagram kolaboracije MVC




U Djangu termini su: model, kontroler je *view*, a view je *template* (MVT).

# Inverzija kontrole (IoC) i princip inverzije zavisnosti (DIP)

- **Loš pristup**: Viša klasa direktno kreira i zavisi od nižih klasa (npr. PostView direktno poziva OpenAIClient).
- **Problem**: Svaka promena u AI modelu zahteva promenu u kodu koji ga koristi.
- **DIP** (**Dependency Inversion Principle**):
  - Moduli višeg nivoa ne bi trebalo da zavise od modula nižeg nivoa. Oba bi trebalo da zavise od apstrakcija.
  - Apstrakcije ne bi trebalo da zavise od detalja. Detalji (implementacija) treba da zavise od apstrakcija.

# Zašto je DIP "spas" u AI projektovanju?

- **Agilnost modela:** Tržište AI modela se menja na nedeljnom nivou (izašao GPT-5, pojeftinio Claude, Llama 3 postala bolja).
- **Prednosti primene DIP-a:**
  - **Hot-swapping:** Zamena provajdera servisa bez izmene biznis logike.
  - **Testabilnost (Mocking):** Možete testirati aplikaciju bez trošenja novca na API pozive, tako što ubrizgate "lažni" (Mock) analizator koji odmah vraća fiksni tekst.
  - **Sigurnost:** Lakša kontrola gde se nalaze API ključevi (samo unutar konkretnog adaptera).

# Praktična implementacija – Dependency Injection (DI)

- **DI u praksi:** Tehnika kojom jedan objekat snabdeva drugi objekat zavisnostima.
- **IoC** (**Inversion of Control**): Umesto da vaša klasa kontroliše zavisnosti, kontrola se predaje spoljašnjem sistemu (kontejneru) koji "ubrizgava" (Inject) potrebne objekte i zna da ih inicijalizuje (jer se klase sa svojim zavisnostima registruju kod njega).

# DI Primer: Injektovanje klijenta za eksterni API

Zamislimo da u Django aplikaciji koristimo eksterni servis za slanje obaveštenja (npr. *NotificationService*):

1. Prvo definišemo *servisnu* klasu koja komunicira sa eksternim API-jem.
2. Django *middleware* instancira tu klasu koristeći podatke (API ključeve) iz *settings.py* i kači je na “request” objekat.
3. Django *view* sada samo pozove servis koji mu je dostavljen, bez znanja o detaljima servisa (api ključevima itd).

# DI Primer: Injektovanje klijenta za eksterni API

```python
# services.py
import requests

class NotificationClient:
    def __init__(self, api_key, url):
        self.api_key = api_key
        self.base_url = url

    def send(self, user_id, message):
        return requests.post(
            f"{self.base_url}/send/",
            json={"user": user_id, "msg": message},
            headers={"Authorization": f"Bearer {self.api_key}"}
        )
```

- Ovoj klasi se injektuje zavisnost api_key u konstruktoru

# DI Primer: Injektovanje klijenta za eksterni API

```python
# middleware.py
from .services import NotificationClient
from django.conf import settings

class NotificationServiceMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # Instanciramo klijent jednom pri pokretanju servera (Singleton stil)
        self.client = NotificationClient(api_key=settings.NOTIF_API_KEY,
                                         url=settings.NOTIF_SERVICE_URL)

    def __call__(self, request):
        # Injektujemo klijent u svaki zahtev
        request.notifications = self.client

        return self.get_response(request)
```

Middleware instancira klijent koristeći ključeve iz settings.py i "kači" ga na request objekat

# DI Primer: Injektovanje klijenta za eksterni API

```
# settings.py

# 1. Definisanje API ključeva i konfiguracije servisa
# Dobra praksa je da se ovi podaci vuku iz .env fajla, ali ovde stoje
direktno radi primera
NOTIF_API_KEY = "sk_live_51Mz..."
NOTIF_SERVICE_URL = "https://obavestenja.rs"

# 2. Registracija middleware-a
MIDDLEWARE = [
     # ostali standardni middleware
    'my_app.middleware.NotificationServiceMiddleware’,
]
```

Django prolazi kroz listu middleware-a definisanu u settings.py i svakom od njih redom "predaje" HTTP zahtev (request) na obradu.Umesto da se u svakom view-u ručno piše kod za nešto, ta informacija injektuje se u request objekat spolja. Kada zahtev konačno stigne do view-a, on već sadrži podatke koje je middleware pripremio.

# DI Primer: Injektovanje klijenta za eksterni API

```
# views.py
def finish_order(request):
    # Logika završetka porudžbine...

    # Koristimo injektovani servis
    request.notifications.send(user_id=request.user.id, message="Vaša
porudžbina je spremna!")

    return HttpResponse("Uspešno!")
```

View sada samo "pozove" servis koji mu je dostavljen, bez znanja o API ključevima.

# Middleware u Djangu – IoC kontejner “za siromašne”

- Zašto LIČI na IoC kontejner?
  - **Inverzija kontrole**: View ne poziva direktno middleware. Django "preuzima kontrolu" i odlučuje kada će koji middleware biti izvršen na osnovu liste u settings.py.
  - **Injektovanje podataka**: Kao što smo videli u primeru, middleware "gura" (injektuje) objekte (poput API klijenta) u request objekat koji zatim stiže do view-a. View samo pasivno koristi ono što mu je "dato".
  - **Konfiguracija**: On koristi centralizovanu konfiguraciju (settings.py) da bi odredio koje će se komponente učitati, što je ključna odlika IoC-a.

# Middleware u Djangu – Zašto NIJE pravi IoC kontejner?

- Pravi IoC kontejner (poput onog u biblioteci *dependency-injector*) je mnogo moćniji:
  - **Upravljanje životnim ciklusom**: Pravi kontejner zna da li treba da napravi novu instancu klase svaki put (Factory) ili da koristi jednu istu (Singleton) za različite delove aplikacije, ne samo za HTTP zahtev.
  - **Granularnost**: Middleware je vezan isključivo za HTTP zahtev-odgovor ciklus. On ne može lako da injektuje zavisnosti u obične Python klase, komande (Management Commands) ili pozadinske taskove (Celery), dok pravi IoC kontejner to može.
  - **Zavisnost o request objektu**: U Django-u si "osuđen" na to da sve što injektuješ moraš da zakačiš na request. U pravom DI sistemu, ti bi zavisnosti mogao da dobiješ direktno u konstruktoru klase ili kao argumente funkcije, bez "prljanja" request objekta.

# Arhitektura vođena događajima

- **Event driven architecture** (**EDA**) je obrazac arhitekture softvera koji naglašava kreiranje, otkrivanje, konzumaciju i reakciju na događaje koji se dešavaju unutar sistema. U EDA, događaji su centralni za arhitekturu, a komponente sistema interaguju tako što proizvode i konzumiraju događaje.
- **Reagovanje na događaje** u aplikaciji može biti sinhrono (da se ceo događaj obradi pre nego što se pređe na obradu sledećeg), ali pravi dobici vide se kada se pravi asinhrono reagovanje na događaje, jer tada aplikcija može da radi druge stvari dok na primer čeka na ulaz/izlaz.

# Sinhrono i asinhrono programiranje

- U ljudskoj komunikaciji **telefonski razgovor** je primer sinhronog komuniciranja, jer se obrada i odziv na ulazne informacije dešava odmah po prijemu bez čekanja.
- **Elektronska pošta** je primer asinhrone komunikacije, jer primalac pošte će pročitati poruku i odgovoriti na nju kada mu to bude zgodno.
- Slično ovome, u asinhronom programiranju, postoji više taskova, gde više taskova može da obavlja različiti posao i ne moraju da čekaju jedan drugog na završetak posla. Pri tome postoji dogovoreni mehanizam gde jedan task može da obavesti drugi da je posao završen i da je rezultat, ako postoji, dostupan.

# Konkurentno i paralelno programiranje

- U kontekstu Django programiranja imamo dve mogućnosti:
- **Asyncio** – konkurentno programiranje korišćenjem kooperativnog multitaskinga: dakle jedan proces/nit (GIL ograničenje) izvršava async taskove.
  - Koristimo za **I/O čekanje**. Dok se čela odgovor od spoljnog API-ja, kod nije zauzet računanjem, već samo čeka rezultat. Asyncio tada dopušta da se obradi drugi zahtev.
- **Celery / Multiprocessing** = Pravi paralelizam (jer pokreće potpuno nove OS procese, od kojih svaki ima svoj GIL i svoje jezgro).
  - Koristimo za **zahtevnu CPU bound obradu** (neko intenzivno lokalno računanje na drugim jezgrima procesora osim onog na kome se služe veb stranice).

# Event-Driven Architecture (EDA) i Message Brokers

- **Koncept:** Sistem komunicira putem **događaja** (Events). Ne pozivamo funkciju direktno, već šaljemo poruku.
- **Ključne komponente:**
  1. **Producer (Django View):** "Korisnik je objavio post."
  2. **Message Broker (Redis/RabbitMQ):** Poštansko sanduče koje čuva poruke.
  3. **Consumer/Worker (Celery):** Pozadinski proces koji uzima poruku i zove spoljni servis.
- **Prednost: Skalabilnost**. Ako imamo hiljade zahteva, samo dodamo još "radnika" (Workera), dok glavni web server ostaje brz i odzivan.

# EDA u praksi – Statusna arhitektura

- **Problem:** Ako server odmah vrati odgovor, kako korisnik dobija rezultat koji AI generiše 10 sekundi kasnije?
- **Rešenja:**
  - **Polling:** Klijent (React) pita server svake 2 sekunde: "Da li je gotovo?".
  - **WebSockets:** Server "gurne" (push) informaciju klijentu čim AI završi posao.
  - **Webhooks:** AI servis obavesti naš server kada je obrada završena (idealno za eksterne AI provajdere).
- **Zlatno pravilo:** Korisničko iskustvo (UX) ne sme da trpi zbog sporosti AI modela.

# EDA u praksi – WebSockets vs WebHooks

| Karakteristika | WebSockets | WebHooks |
|---|---|---|
| **Trajanje veze** | Perzistentna (stalno otvorena) | Kratkotrajna (otvori se, pošalje, zatvori) |
| **Smer podataka** | Dvosmerno (Server <-> Klijent) | Jednosmerno (Server -> Klijent) |
| **Protokol** | WS / WSS (baziran na TCP-u) | Standardni HTTP / HTTPS |
| **Resursi** | Zahtevni za server (mnogo otvorenih veza) | Lagani (svaki poziv je nezavisan) |

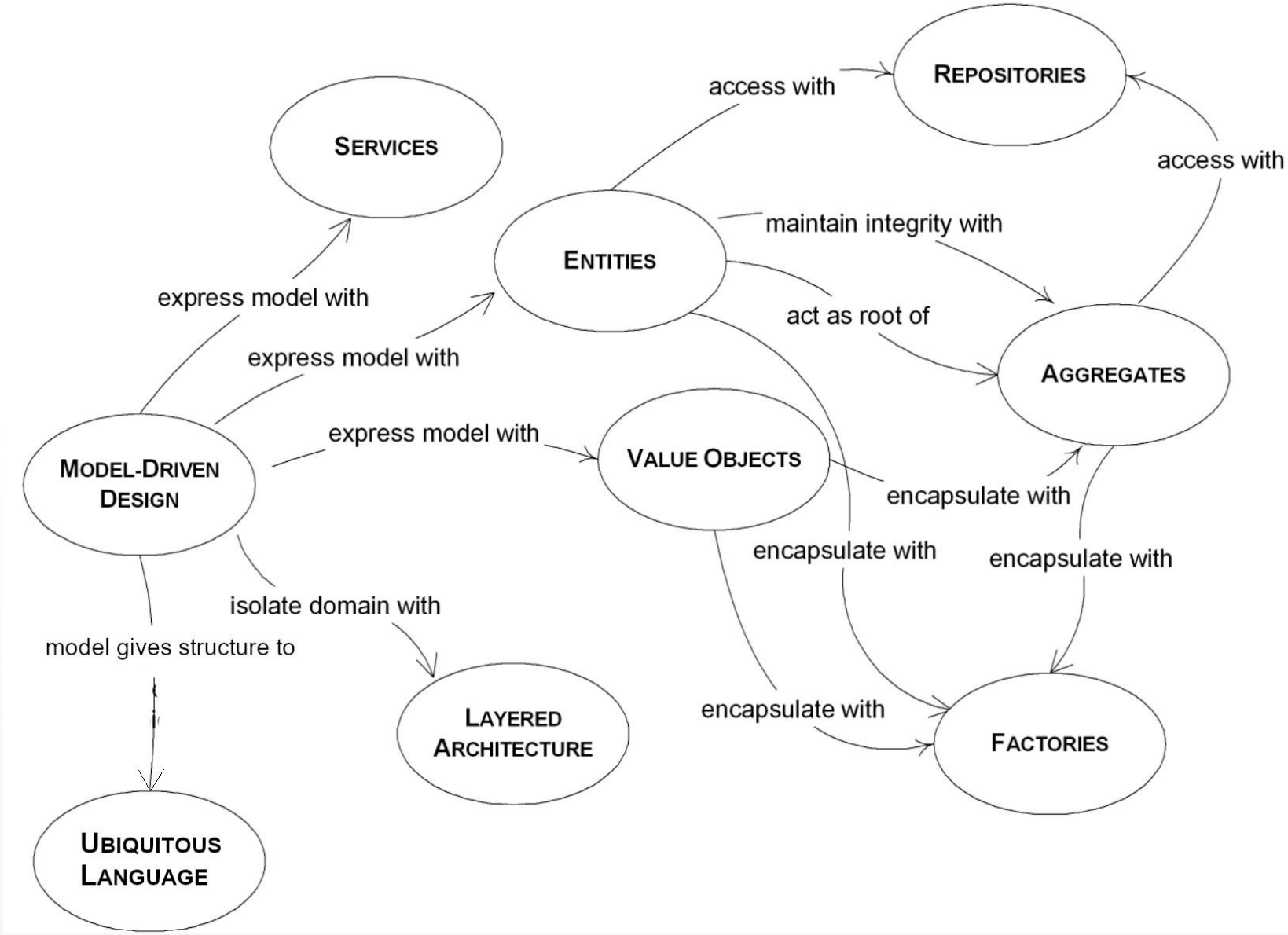
Kada šta koristiti?Ako ti treba da dvoje ljudi kuca u isto vreme u istom dokumentu, koristi WebSockets. Ako samo želiš da tvoj server dobije info kada ti neko uplati novac na račun, WebHooks su pravo rešenje.

# Dizajn vođen domenom (Domain driven design, DDD)

# Dizajn vođen domenom

- Dizajn vođen domenom (DDD) je metodologija dizajna koja se fokusira na izgradnju softvera koji odražava domen iz stvarnog sveta kojem služi. To uključuje razumevanje domena i njegovo modelovanje u kodu, stvaranje zajedničkog jezika između programera i zainteresovanih strana i korišćenje tog modela za pokretanje dizajna softvera.
- DDD se dakle odnosi kako na strukturu same aplikacije, tako i na metodologiju projektovanja aplikacije
- Referenca: Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, ISBN: 0321125215
- Preporučeni sažetak ove knjige: https://www.infoq.com/minibooks/domain-driven-design-quickly/

# Ključni koncepti DDD-a



# Ključni koncepti DDD-a

- *Ubiquitous language*, odnosno zajednički jezik.
- *Entiteti (imaju identitet)* i *value objekti (definisani vrednostima)*
  - modeluju realne poslovne koncepte.
- *Agregati*, logička grupa objekata (uz očuvanje poslovnih pravila)
- Factory-ji koji kreiraju objekte i agregate i
- *Repozitorijumi*, koji služe za pristup tim agregatima (njihovo čuvanje u bazi).
- *Servisi* orkestriraju interakcije (poslovnu logiku) između ovih objekata.

# Slojevi u DDD arhitekturi

- Sistem se organizuje u nekoliko slojeva:
  - Prezentacioni sloj – bavi se interakcijom sa korisnikom, prima zahtev i poziva aplikacione servise
  - Aplikacioni sloj – orkestrira operacije, ali ne sadrži poslovna pravila
  - Domenski sloj – poslovna logika (izolovan i nezavisan od tehničkih detalja – http, json, baza,…)
  - Infrastrukturni sloj – tehnički detalji za podršku ostalim slojevima: baza, pozivanje spoljnih APIja,…

# Primer: aplikacija za naručivanje proizvoda

- Struktura DDD projekta realizovanog sa django frejmvorkom:

```
project/
├── orders/
│   ├── domain/
│   │   ├── models.py
│   │   ├── value_objects.py
│   │   └── repositories.py
│   │
│   ├── application/
│   │   └── services.py
│   │
│   ├── infrastructure/
│   │   └── repositories.py
│   │
│   ├── presentation/
│   │   └── views.py
│   │
│   └── models.py   # Django ORM (infrastructure zapravo)
```

# Primer – domenski sloj

**Entitet: Order**

```
class Order:
 def __init__(self, id, items=None, status="CREATED"):
        self.id = id
        self.items = items or []
        self.status = status

    def add_item(self, product_id, price, quantity):
        self.items.append({
            "product_id": product_id,
            "price": price,
            "quantity": quantity
        })

    def total(self):
        return sum(i["price"] * i["quantity"] for i in self.items)

    def confirm(self):
        if not self.items:
            raise ValueError("Ord. must have at least one item")
        self.status = "CONFIRMED„
```

**Value Object: Money**

```
class Money:
    def __init__(self, amount, currency):
        self.amount = amount
        self.currency = currency

    def add(self, other):
        if self.currency != other.currency:
            raise ValueError("Currency mismatch")
        return Money(self.amount + other.amount,
                     self.currency)
```

**Repository interfejs**

```
class OrderRepository:
    def save(self, order):
        raise NotImplementedError

    def find_by_id(self, order_id):
        raise NotImplementedError
```

# Primer – Aplikativni sloj

**Use-case servis**

```
from orders.domain.models import Order

class OrderService:
    def __init__(self, order_repository):
        self.order_repository = order_repository

    def create_order(self, order_id):
        order = Order(order_id)
        self.order_repository.save(order)
        return order

    def add_item(self, order_id, product_id, price,
quantity):
        order = self.order_repository.find_by_id(order_id)
        order.add_item(product_id, price, quantity)
        self.order_repository.save(order)

    def confirm_order(self, order_id):
        order = self.order_repository.find_by_id(order_id)
        order.confirm()
        self.order_repository.save(order)
```

# Primer – infrastrukturni sloj (Django ORM)

**Django modeli (persistencija)**

```
from django.db import models

class OrderModel(models.Model):
    id = models.CharField(primary_key=True, max_length=50)
    status = models.CharField(max_length=20)

class OrderItemModel(models.Model):
    order = models.ForeignKey(OrderModel, related_name="items", on_delete=models.CASCADE)
    product_id = models.CharField(max_length=50)
    price = models.FloatField()
    quantity = models.IntegerField()
```

# Primer – infrastructure sloj (Django ORM)

**Repository implementacija**

```
from orders.domain.models import Order
from orders.domain.repositories import OrderRepository
from orders.models import OrderModel, OrderItemModel

class DjangoOrderRepository(OrderRepository):
    def save(self, order):
        order_model, _ =
            OrderModel.objects.get_or_create(id=order.id)
        order_model.status = order.status
        order_model.save()
        # obriši stare stavke i upiši nove
        OrderItemModel.objects.filter(order=order_model)
                                        .delete()
        for item in order.items:
            OrderItemModel.objects.create(
                order=order_model,
                product_id=item["product_id"],
                price=item["price"],
                quantity=item["quantity"]
            )

    def find_by_id(self, order_id):
        order_model = OrderModel.objects.get(id=order_id)
        items = []
        for i in order_model.items.all():
            items.append({
                "product_id": i.product_id,
                "price": i.price,
                "quantity": i.quantity
            })
        return Order(
            id=order_model.id,
            items=items,
            status=order_model.status
        )
```

# Primer – prezentacioni sloj (Django pogledi)

```
import json
from django.http import JsonResponse
from django.views import View

from orders.application.services import OrderService
from orders.infrastructure.repositories import
    DjangoOrderRepository

repo = DjangoOrderRepository()
service = OrderService(repo)

class OrderView(View):
  def post(self, request):
    data = json.loads(request.body)
    order = service.create_order(data["id"])
    return JsonResponse({"id": order.id, "status":
order.status})
```

```
class AddItemView(View):
  def post(self, request, order_id):
    data = json.loads(request.body)
    service.add_item(
      order_id,
      data["product_id"],
      data["price"],
      data["quantity"]
    )
    return JsonResponse({"status": "ok"})


class ConfirmOrderView(View):
  def post(self, request, order_id):
    service.confirm_order(order_id)
    return JsonResponse({"status": "confirmed"})
```

# Šta je ovde u skladu sa DDD

- Domen (Order) je čist Python — nema Django importa
- ORM modeli su u infrastructure sloju
- Repository prevodi između ORM i domena
- View ne sadrži poslovnu logiku

## Tipične greške u Django+DDD:

1. Stavljanje logike direktno u models.Model
2. Mešanje ORM i domen logike
3. “Fat views” umesto application servisa
4. Preskakanje repository sloja

# Ostali važni pojmovi u DDD

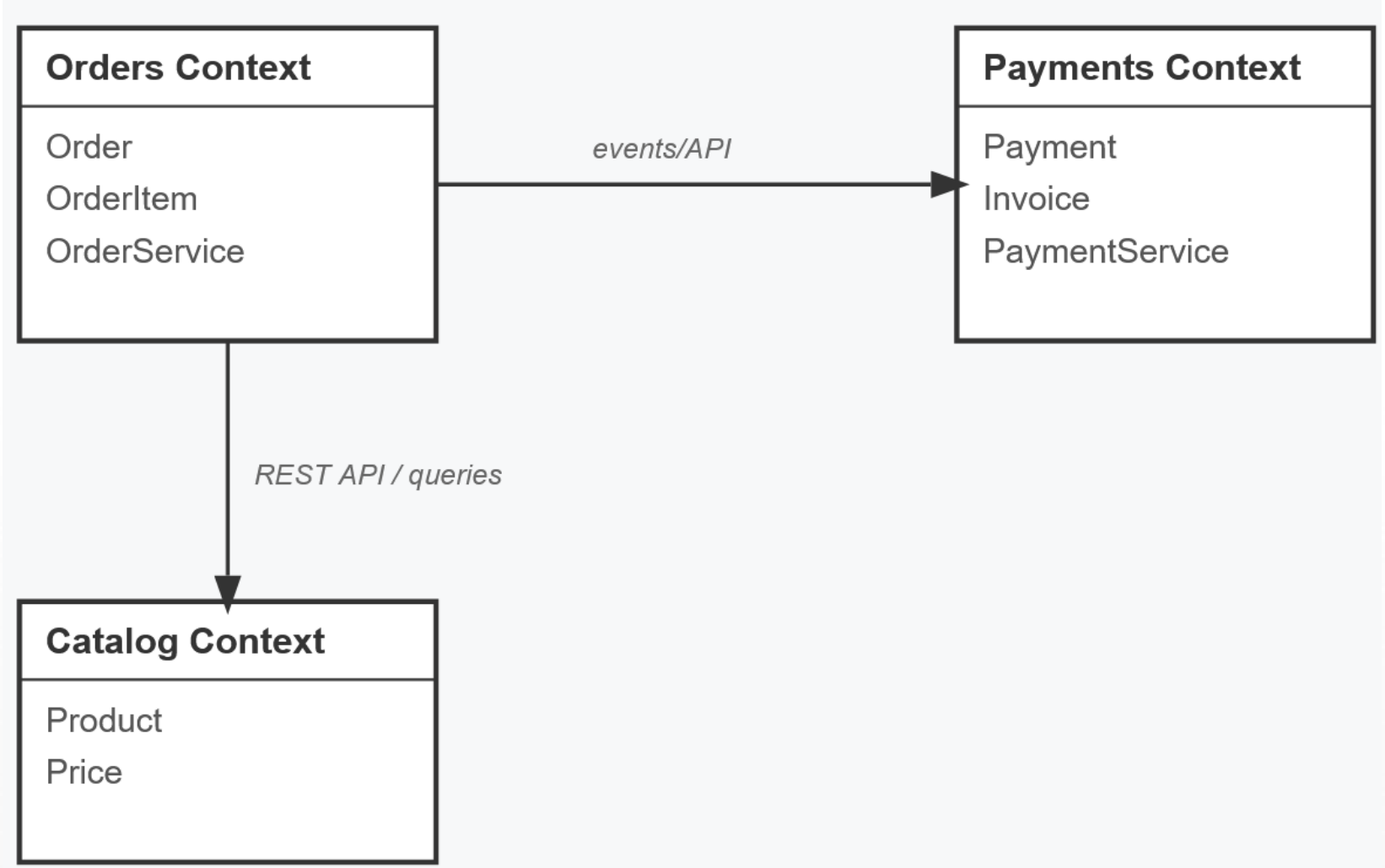
- **Ograničeni kontekst** (**Bounded context**) => logička granica unutar koje određeni domenski model ima jednoznačno značenje
- Unutar jednog konteksta:
  - termini imaju tačno definisano značenje
  - pravila i ponašanje su konzistentni
- Između konteksta:
  - modeli se ne mešaju direktno
  - koristi se mapiranje (**context map**) koje prikazuje kako konteksti međusobno komuniciraju

# Ostali važni pojmovi u DDD

## Ključna ideja Context Map-a

- Bounded contexts su:
  - nezavisni modeli (implementirani posebnim veb aplikacijama ili mikroservisima)
  - ali povezani preko eksplicitnih ugovora
  - (npr. putem API-ja, dogadjaja ili čak **Anti-Corruption Layera** (**ACL**), koji prevodi i izoluje eksterni model da ne “zagadi” lokalni domen).

# Primer mapiranja konteksta



# Primer mapiranja konteksta

- **Orders Context**
  - Glavni domen: porudžbine
  - Ne zna ništa o plaćanju ili katalogu interno
  - Komunicira spolja
- **2. Payments Context**
  - Odgovoran za:
    - naplatu
    - transakcije
    - fakture
  - Ne zna detalje porudžbine, samo prima “signal” da treba platiti
    - tipična komunikacija:
    - event: OrderConfirmed
    - ili API poziv: “naplati orderId”
- **3. Catalog Context**
  - Drži proizvode i cene
  - Orders ga koristi za:
    - prikaz proizvoda
    - proveru cena (ili snapshot)

# Prednosti i mane DDD-a

- PREDNOSTI:
  - Bolje razumevanje problema
  - Jasnija komunikacija u timu
  - Kod bolje odražava poslovnu logiku => lakše se održava
- MANE:
  - Dodatna kompleksnost
  - Zahteva disciplinu i iskustvo

# Kada (ne)koristiti DDD

- Za kompleksan domen, veliki razvojni tim i dugoročan razvoj sistema
- Za male i jednostavne aplikacije često je previše težak i nepotreban

# Rezime predavanja

1. **Pristup**: Django vs React (CSR vs SSR).
2. **Struktura**: Kako SOLID, MVC, DDD i N-Tier čuvaju red u sistemu.
3. **Strategija**: Kako pomoću IoC/DIP lako menjati API implementacije.
4. **Performanse**: Kako EDA rešava problem sporosti AI-a.

# Zaključak

- **Arhitektura kao Prompt**:  Šabloni (MVC, DIP, IOC, EDA) se koriste da bi AI-u dali precizne instrukcije.
- **Human-in-the-loop**: Arhitektura mora predvideti sloj gde čovek odobrava ono što je AI uradio pre nego što to postane "javno".
- "Dobar programer piše kod koji radi. Vrhunski softverski inženjer projektuje arhitekturu u kojoj AI može bezbedno da radi za njega."