

# Kazalo vsebine

---

- Kazalo vsebine
- Opravljanje predmeta
- Razvoj IT arhitektur
  - IT agilnost podjetji
  - IT arhitektura
    - Software platforma
    - Namestitev aplikacije
    - Vrste SQL baz
      - ACID baze
      - BASE baze
  - Upravljanje z odvisnostmi
- Maven
  - Prednosti uporabe Maven
  - POM - Project Object Model
    - Primer POM datoteke
    - Dedovanje POM datotek
    - Agregacija modulov
  - Maven življenjski cikli
  - Struktura Maven projekta
  - Prakse pri uporabi Maven
    - Slaba praksa: podvajanje odvisnosti
    - DObra praksa: uporaba binarnega repozitorija
  - Izdelava dokumentacije
- JDBC - Java Database Connectivity
  - Tipi JDBC gonilnikov
    - Tip 1: JDBC-ODBC most
    - Tip 2: delni javanski gonilnik
    - Tip 3: javanski/mrežni gonilnik
    - Tip 4: čisti javanski gonilnik
  - Koraki pri uporabi JDBC
    - Korak 1: Nalaganje gonilnika
    - Korak 2: Sestavljenje URL niza za povezavo na bazo
    - Korak 3: Vzpostavljanje povezave
    - Korak 4: Kreiranje objekta `Statement`, `PreparedStatement` ali `CallableStatement`
      - `Statement`
      - `PreparedStatement`
      - `CallableStatement`
    - Korak 5: Izvršitev SQL povpraševanj ali shranjenih procedur
      - Uporaba objekta `Statement`
      - Uporaba objekta `PreparedStatement`
    - Korak 6: Obdelava rezultatov
    - Korak 7: Zapiranje povezave
  - JDBC transakcije

- JDBC Connection Pool
- Dobre prakse uporabe JDBC
- DAO (Data Access Object)
  - Vzorec DAO
  - Naloge DAO
  - Generiranje baznega DAO
- Java Persistence API (JPA)
  - Anotiranje entitetnih razredov
    - `@SecondaryTable`
    - `@Id`
    - `@Temporal` "Kalendar"
    - `@Transient`
    - `@Enumerated`
    - `@ElementCollection` in `@CollectionTable`
  - Relacije med entitetami
  - Dedovanje
  - Struktura Java Persistence APIja
- Nivo poslovne logike in CDI
  - CDI - Contexts and Dependency Injection
    - CDI zrna
  - CDI container
  - Interceptor
    - `GET` za branje vira
      - `GET` za branje določenega vira
      - `GET` na viru
    - `POST` za ustvarjanje
    - `PUT` za posodabljanje
    - `DELETE` za brisanje
  - Format sporočil
    - JSON
      - JSON header

## Opravljanje predmeta

---

- 50% projekt pri vajah
- 50% kolokviji/izpit

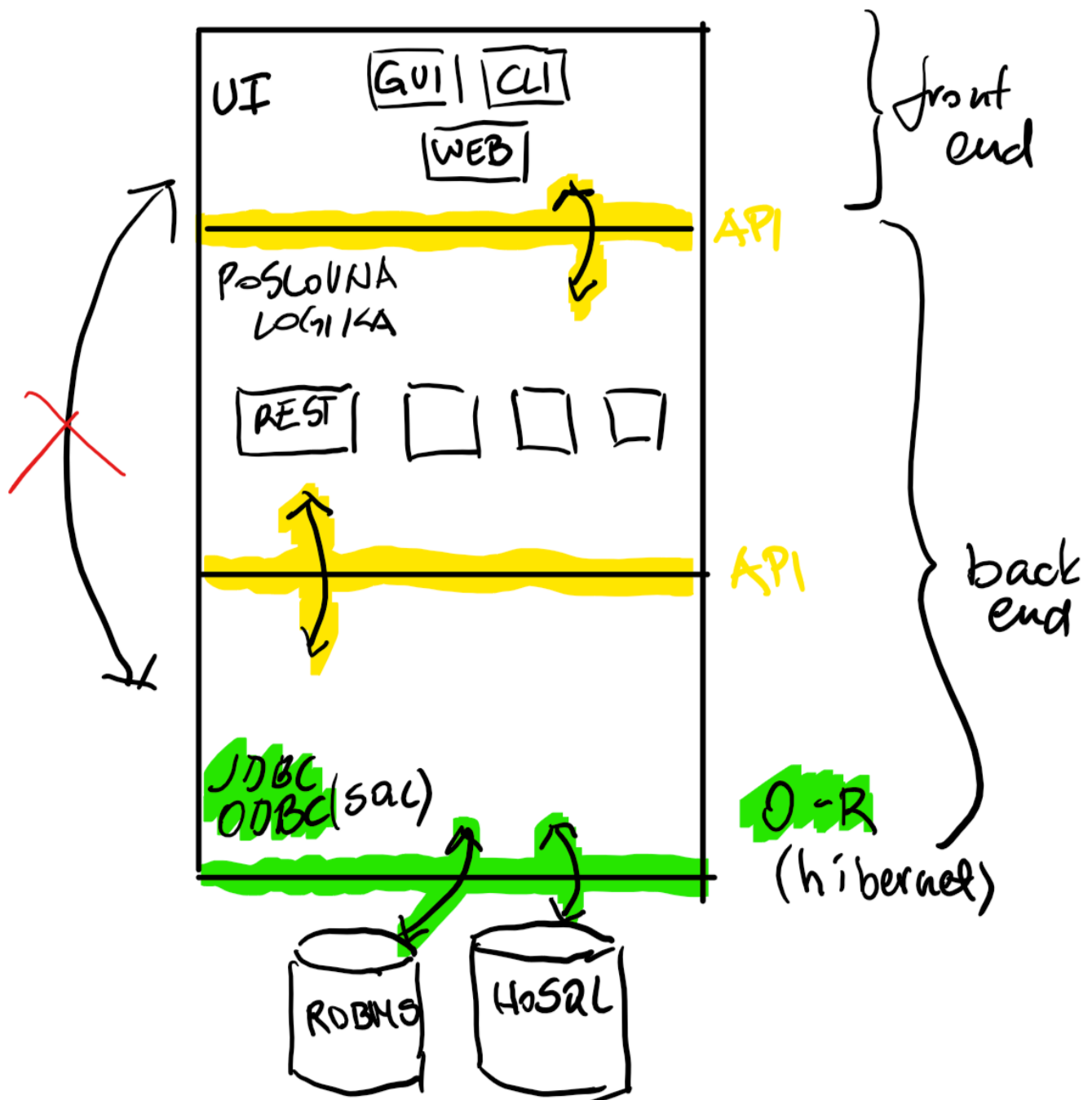
### Kolokviji

v času predavanj (**oba** nad 50% + **povprečje** nad 50%)

1. kolokvij: 18.11.2019
2. kolokvij: 20.01.2020

"Več znate, več zaslužite" Jurič

## Razvoj IT arhitektur



Vrste uporabniškega vmesnika:

- CLI - uporabniški vmesnik v konzoli
- GUI - grafični vmesnik, program moramo prej namestiti na končno napravo (Word,...), grafični elementi operacijskega elementa
- Web - vmesnik se izvaja v browser-ju
  - Server side web development SSWD
    - na client se prenese (večinoma) čisti HTML
    - vsaka sprememba se mora ponovno prenesti s strežnika
  - Client side web development CSWD (SPA - single page application)
    - izvajanje na clientu (JS)

IT agilnost podjetji

### 1. **Doba centralnih računalnikov**

- batch/množične obdelave (pripis obresti 1x mesečno, plače 1x mesečno,...), velika količina podatkov 1x v *časovne obdobju*
- OLTP online transakcije

### 2. **Dvo-nivojska arhitektura** - strežnik odjemalec (*fat client* - na klientu je veliko podatkov, na strežniku samo podatki (baza))

### 3. **Tro-nivojska arhitektura** (internet) - ločujemo front, backend in podatke

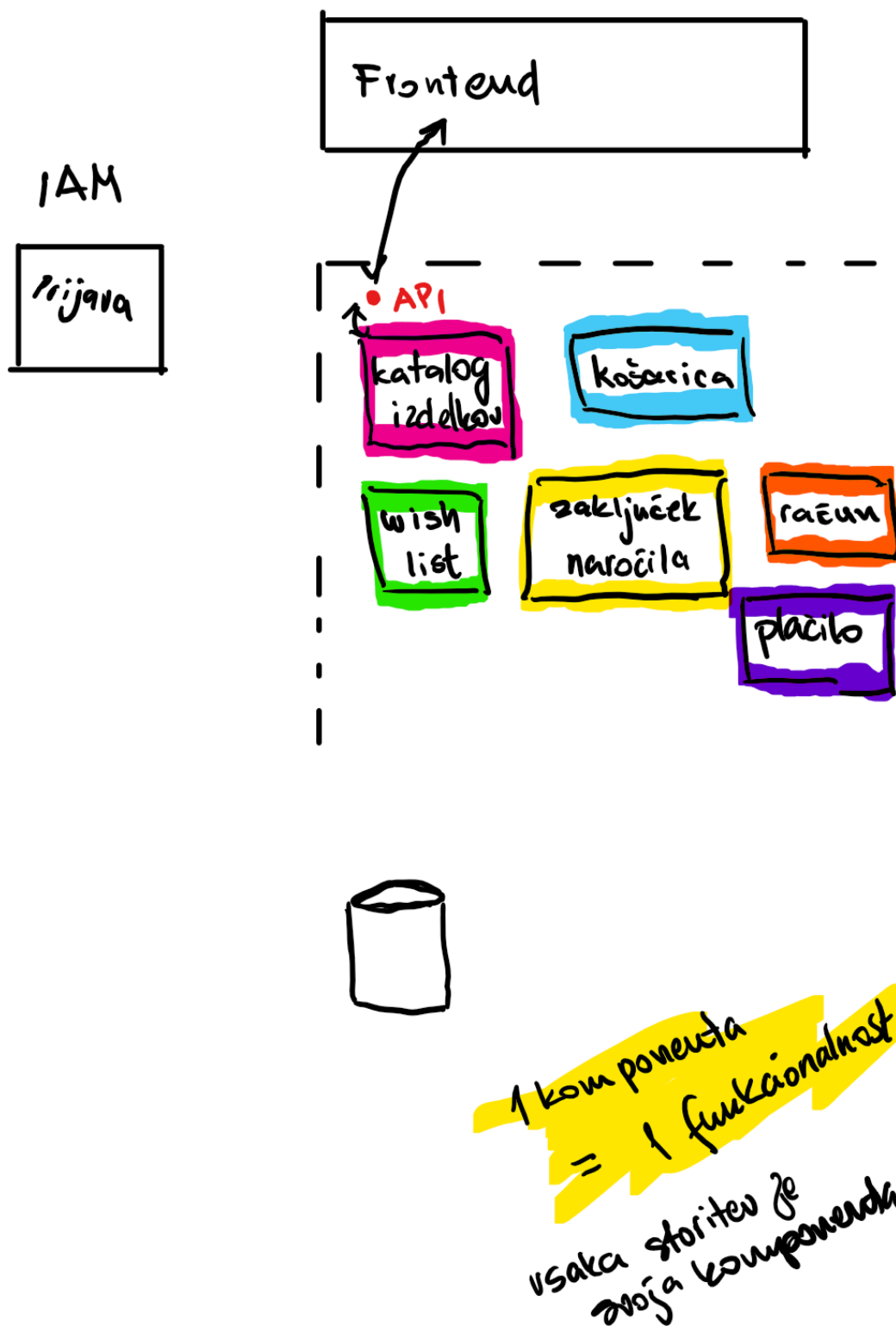
- premik iz razmišljanja o enotni aplikaciji v aplikacijo kot sestavljeno iz različnih delov

### 4. **SOA - server oriented application** - aplikacija je sestavljena iz množice storitev (aplikacijo sestavimo iz "sestavnih delov")

### 5. **Mikrostoritve**

- *Iz katerih sestavnih delov naj bo sestavljena aplikacija?*

## ■ Spletna trgovina



### ○ Prednosti

- večja preglednost izvirne kode
- manjša možnost sesutja celotne aplikacije (bolj robustna)
- hitrejša (boljša možnost optimizacije)
- skalabilnost (poganjanje instanc posameznih komponent) - FE se ne skalira, v BE skaliramo podatke z uporabo NoSQL (SQL se ne skalirajo tako dobro zaradi *cap teorema*)

### ○ Slabosti

- API - komunikacija med komponentami (ozko grlo)

## IT arhitektura

### Software platforma

Skupek strežnikov storitev, ki jih za poganjanje aplikacije potrebujemo:

- web server
- file server
- database server (DBMS)
- IAM - identity server (npr. OAuth2)

### Namestitev aplikacije

**Monolitna** namestitev - aplikacijo namestimo v končni sistem v enem kosu **Mikrostoritvena** namestitev - vsako komponento posebej namestimo (npr. z uporabo Dockerja)

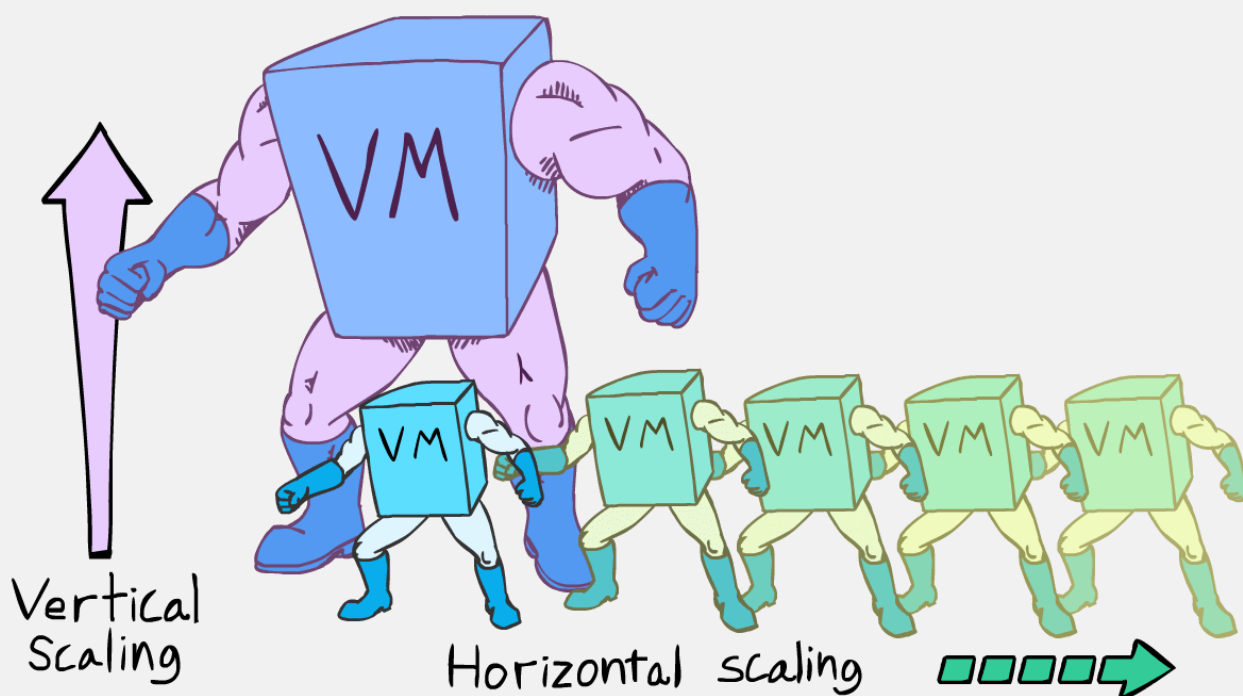
Über

Jurič

Java EE

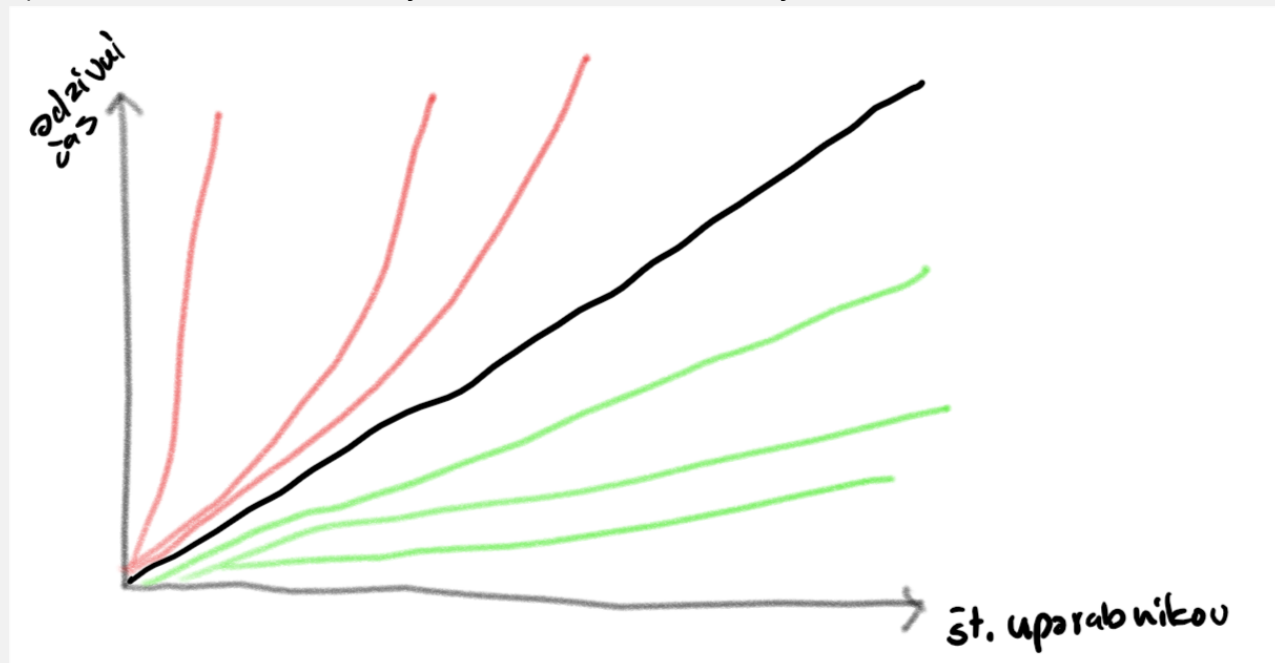
vrsta	datoteke
Java SE	JAR
Java EE zdaj Jakarta EE	WAR, EAR

### Skalabilnost



Skalabilnost nam pove kaj se bo zgodilo z odzivnim časom, ko povečujemo število aktivnih

uporabnikov. Težimo k temu, da je skalabilnost sistema čim boljša.



### Vertikalna skalabilnost

Povečujemo strojne zmogljivosti aplikacije (CPU, RAM, network)

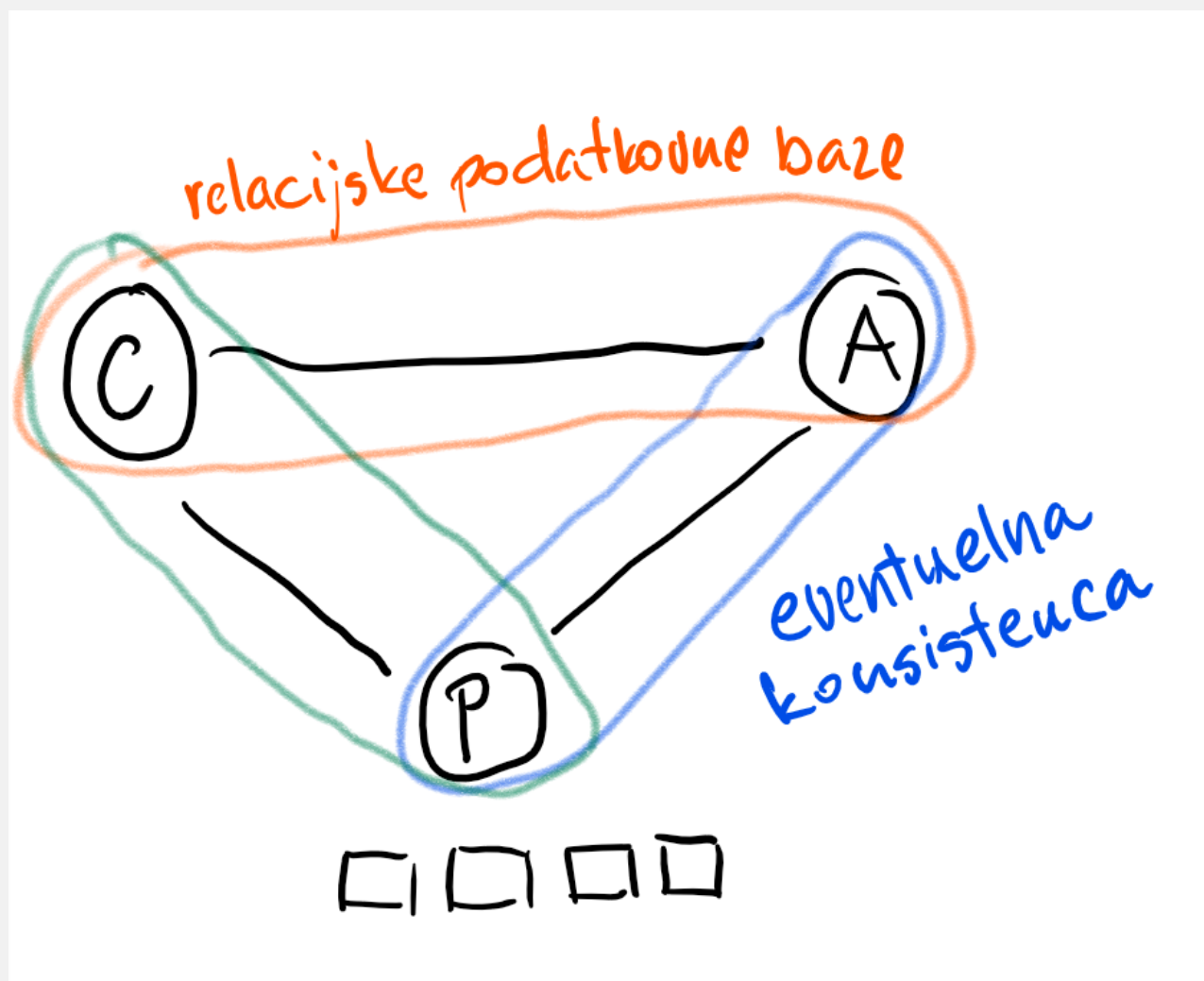
### Horizontalna skalabilnost

Zmožnost poganjanja večih instanc posamezne komponente (potreno pravilno programiranje)

### Cap theorem

C - transakcijska konsistentnost podatkov A - razpoložljivost podatkov v konsistentni obliki v vseh vozliščih P - particijska enostavnost: enostavnost horizontalne skalabilnosti izbereš lahko največ dve od

treh



"Če zaradi tega vržete puško v koruzo in ne govorite z Metko, je dobro, da hodite na ta predavanja" Jurič, 21.10.19

## Vrste SQL baz

### ACID baze

**A**tomicity, **C**onsistency, **I**solation, **D**urability (Atomarnost, Konsistentnos, Izolacija, ) Običajno SQL relacijske baze

### BASE baze

Pomembnejša je razpoložljivost od razpoložljivosti

## Upravljanje z odvisnostmi

Uporabljamo za avtomatizacijo rešije:

- upravljanje z odvisnostmi
- build cycle - priprava za namestitev aplikacije Primeri: *Maven*, *Gradle*. Project Object Model (POM)

## Okolje aplikacije



1. **Razvojno okolje** (DEV)
2. **Testno okolje** - testira QA
3. **User testno okolje** - še eno testno okolje, testirajo končni uporabniki
4. **Produksijsko okolje** - okolje, kjer se izvaja aplikacija

## Maven

### Verzioriranje

**a.b.c-okolje** (npr. 1.0.0)

- *a* - major
- *b* - minor
- *c* - release
- *okolje* - SNAPSHOT (testno okolje)

### Prednosti uporabe Maven

- boljša vidnost in transparentnost razvojnega procesa
- apliciranje splošno sprejetih dobrih praks (verzioriranje)
- standardizacija (enotna struktura projektov)
- upravljanje z odvisnostmi
- samodejno generiranje spletne strani in dokumentacije

### POM - Project Object Model

V datoteki **pom.xml** definiramo:

lastnost	definicija
naziv projekta	<code>&lt;name&gt;</code>
verzija	<code>&lt;version&gt;</code>
odvisnosti	<code>&lt;dependencies&gt;&lt;dependency&gt;</code>
cilji (goals)	
vtičniki (plugins)	
metapodatki	

- V **pom.xml** datoteki so obvezni podatki **groupId**, **artifactId**, **version** in **modelVersion**.
- Lahko uporabljamo koncept dedovanja
- Vsak **pom.xml** deduje od super POM-a

### Primer POM datoteke

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
```

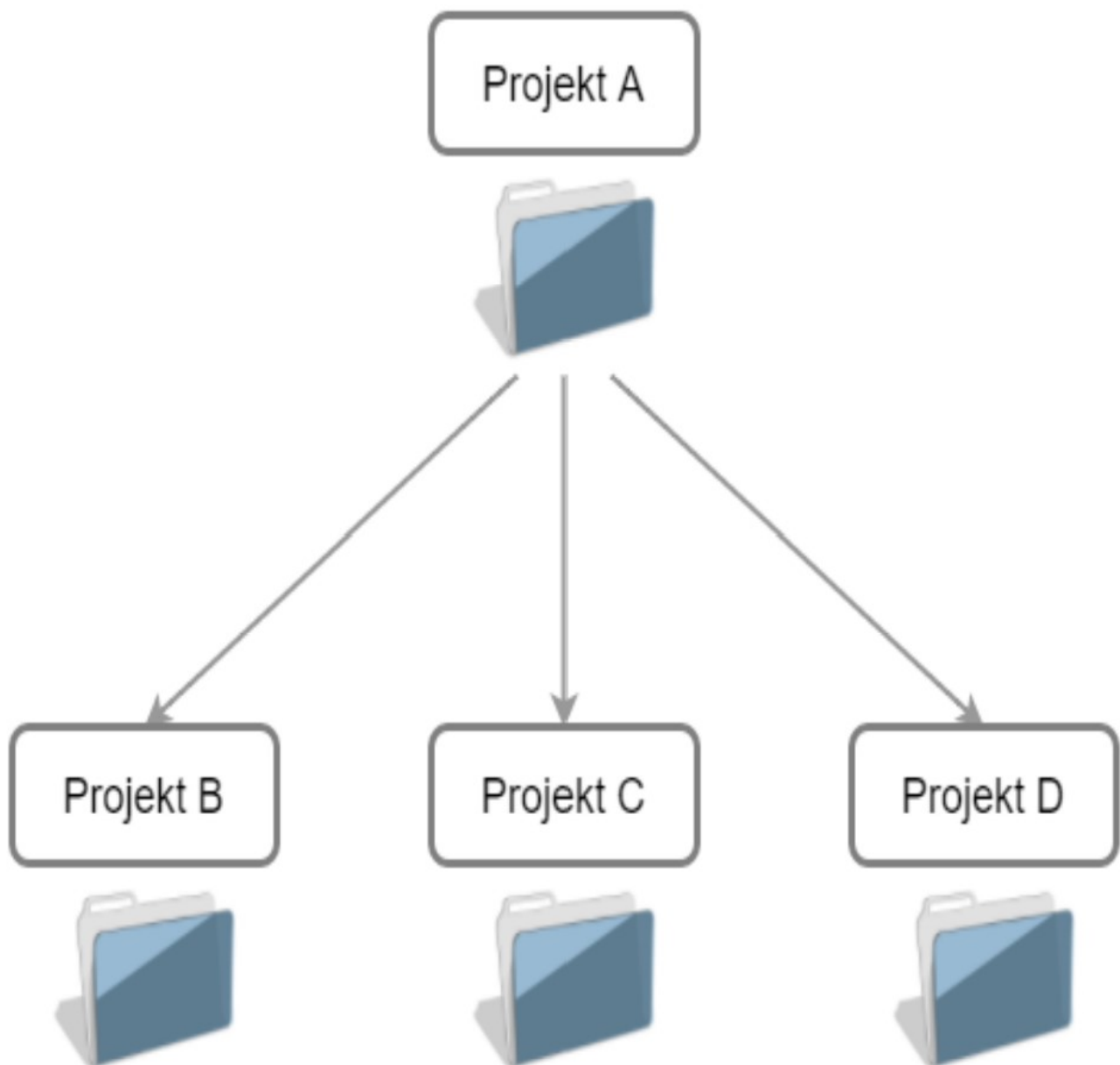
```
<modelVersion>4.0.0</modelVersion>
<groupId>com.demo</groupId>
<artifactId>projektA</artifactId>

<!-- privzeto jar (izpuščamo), ostale: war, ejb, rar, ear, pom, custom-->
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit </artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

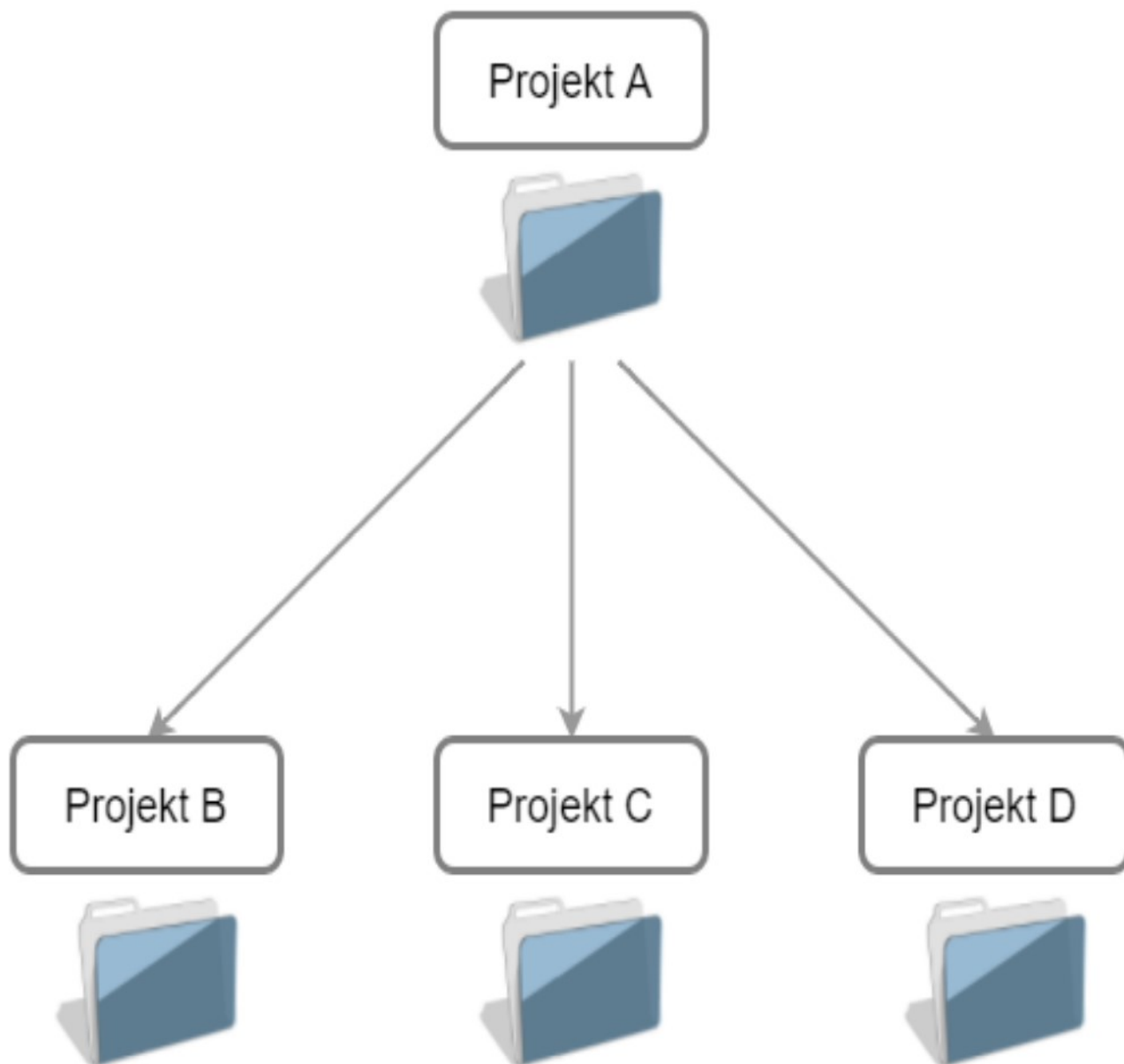
Ustvari artefakt *projektA-0.0.1-SNAPSHOT.war*

Dedovanje POM datotek



*Projekti B, C, D dedujejo verzijo, groupId, način pakiranja in vse druge odvisnosti in konfiguracijo vtičnikov, če nimajo sami drugače definirano.*

Agregacija modulov



*Vsi ukazi nad A se izvedejo tudi nad B, C in D.*

## Maven življenjski cikli

- **default** - namenjen korakom buildanja in nameščanja, najpomembnejši cilji (izvedejo se tudi vse predhodne faze):
  - validate
  - compile (jar file iz artifactID in verzije)
  - package - naredil compile in sestavi jar file
- **clean** - čiščenje za seboj (predhodne builde)
  - pre-clean
  - clean
  - post-clean
- **site** (oblikovanje dokumentacije)
  - pre-site
  - site
  - post-site

- site-deploy

Building iz komandne vrstice:

```
mvn clean
//--

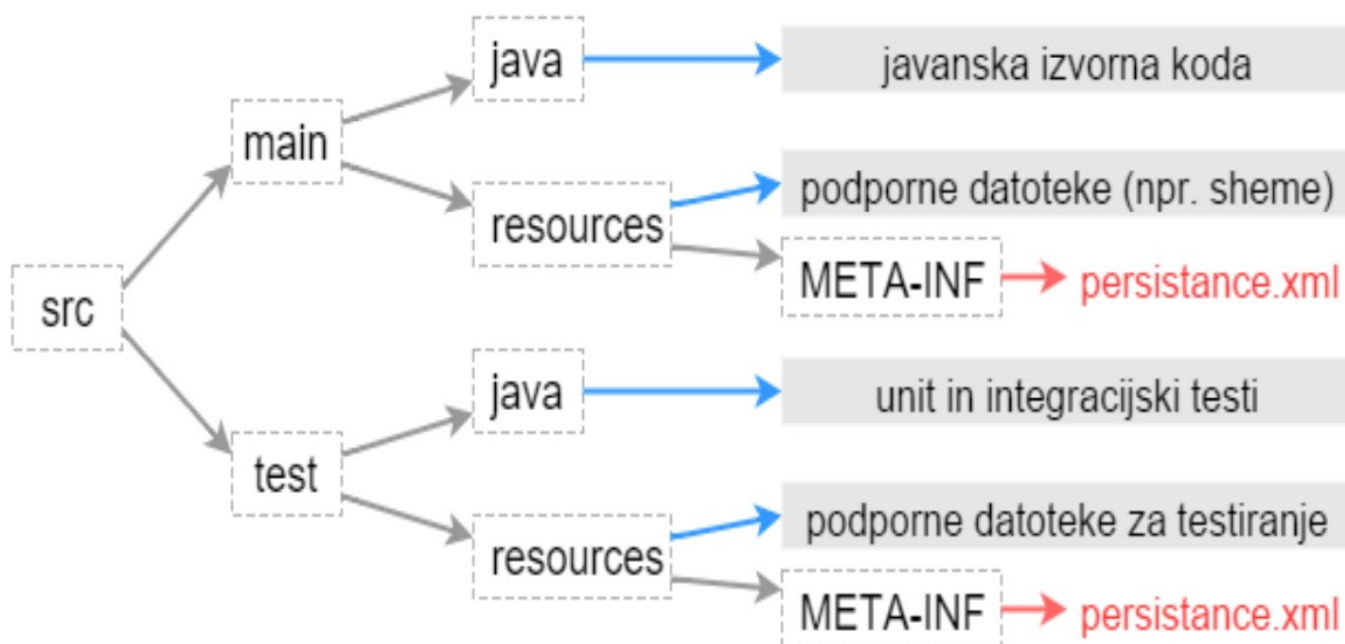
// dobimo .class file
mvn compile
//--

// compile & .jar file
mvn package
//--

mvn run

//kombinacija dveh ciljev
mvn clean package
```

## Struktura Maven projekta



## Prakse pri uporabi Maven

Slaba praksa: podvajanje odvisnosti

Podvajanje odvisnosti za vsak projekt (npr. kopiranje v mapo lib).

DObra praksa: uporaba binarnega repozitorija

**Repozitorij** je skupna lokacija za vse odvisnosti projektov, prednosti:

- obstaja samo ena kopija
- odvisnosti so shranjene izven projekta
- odvisnosti so definirane v `pom.xml` Privzeti oddaljen repozitorij je **maven control** (*repo1.maven.org*), uporabljamo lahko tudi druge. **Organizacijski repozitorij** hrani vse artefakte, ki izboljšujejo varnost in hitrost. **Lokalni repozitorij** predstavlja predpomnilnik za artefakte iz oddaljenih repozitorijev.

## Izdelava dokumentacije

Spletno stran z dokumentacijo generiramo z uporabo ukaza `mvn site`, spletne strani se nahajajo v mapi `target/site`.

## JDBC - Java Database Connectivity

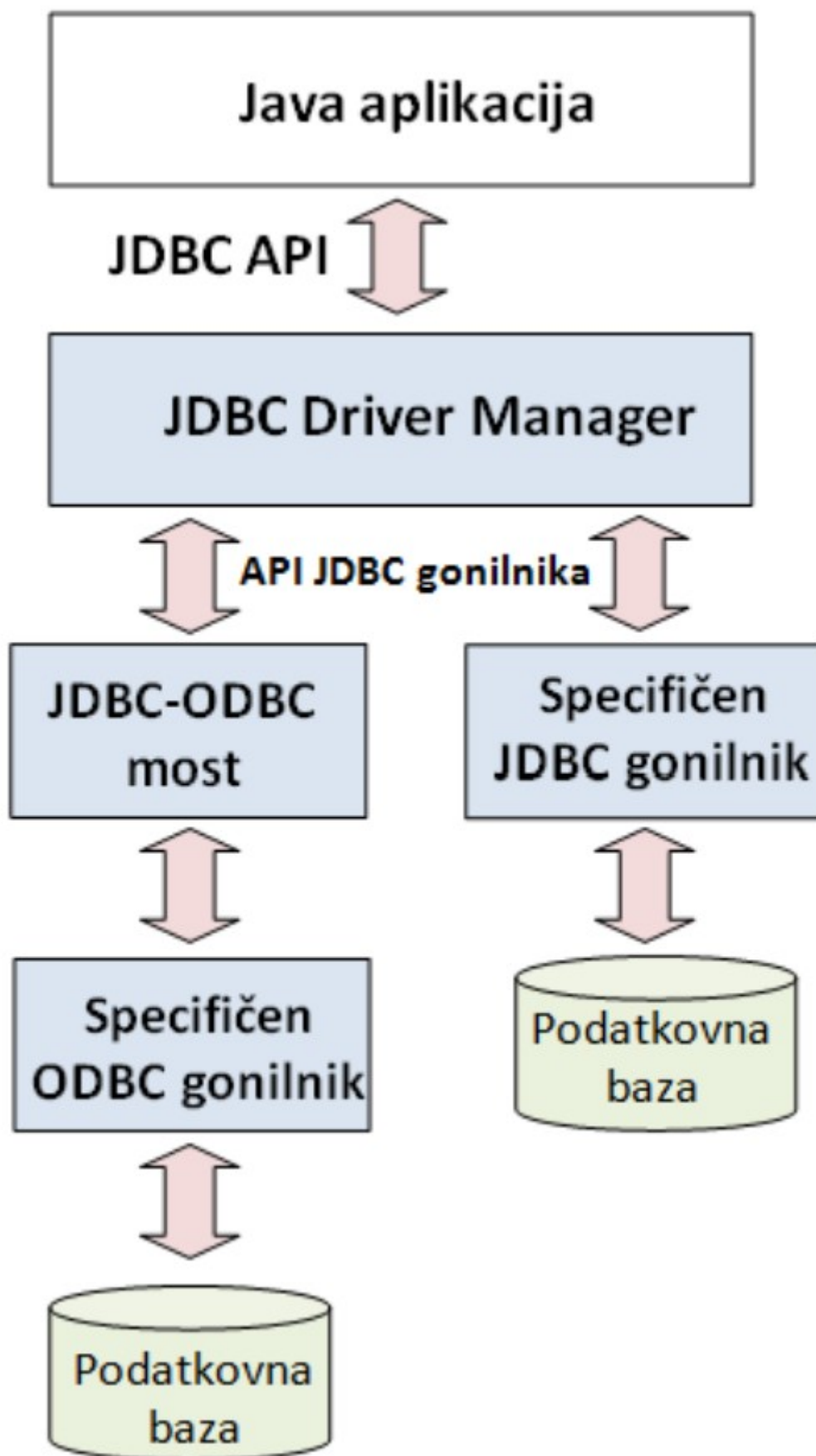
---

Standardna javanska knjižnica JDBC API standarizira:

- vzpostavljanje povezave na bazo
- izvajanje SQL povpraševanj
- strukturo rezultatov povpraševanj

JDBC sestavljajo:

- JDBC API - čisti javanski API
- JDBC Driver Manager ki komunicira s produktno-specifinimi gonilniki, ki opravijo dejansko komunikacijo s podatkovno bazo

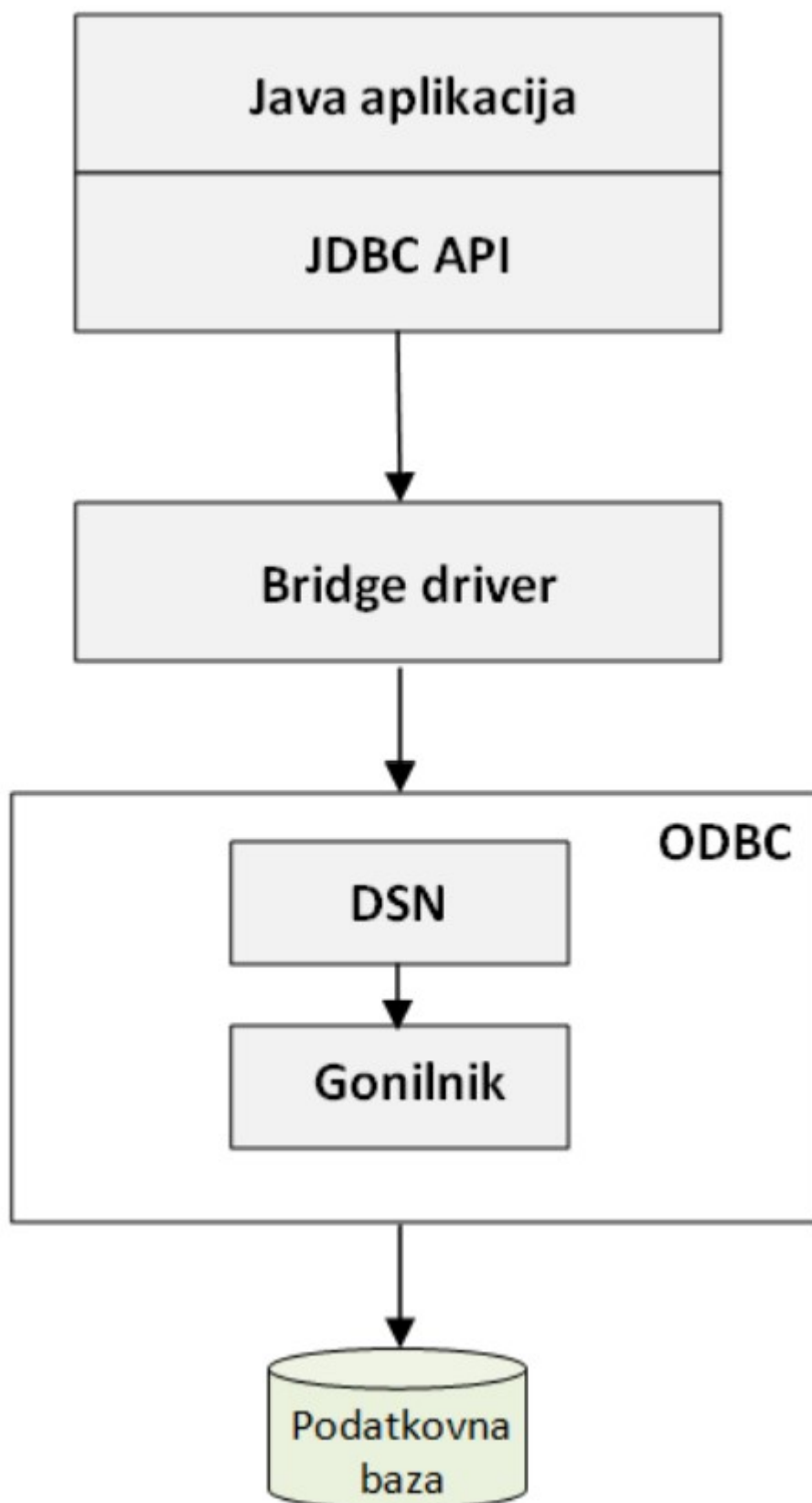


## Tipi JDBC gonilnikov

### Tip 1: JDBC-ODBC most

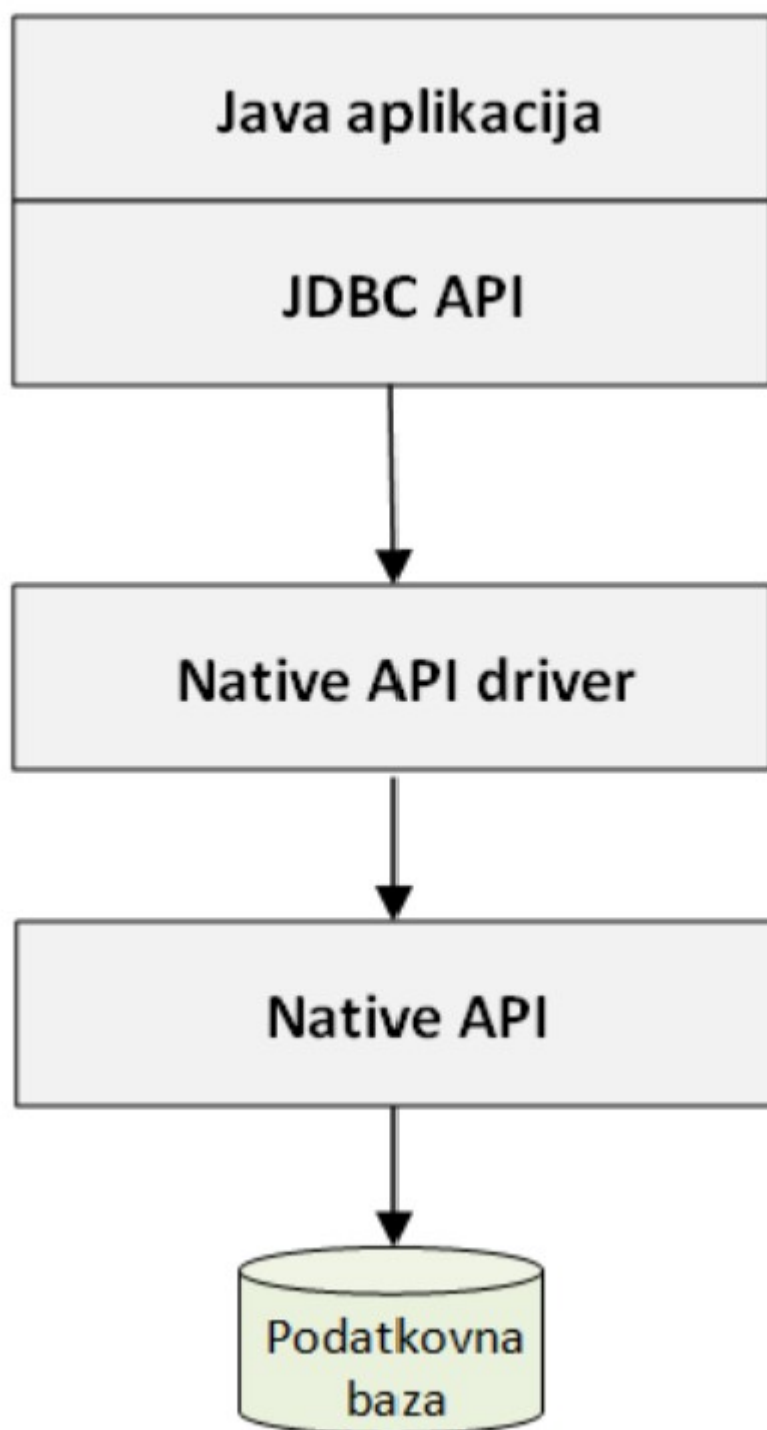
Vsi JDBC klici se pretvorijo v ODBC klice in jih kot take posredujejo ODBC gonilniku, ki je generičen API za dostop do baze. Zaradi slabe prenosljivosti in slabega performansa je primerna samo za testne namene ali kadar ni na voljo javanskega gonilnika.





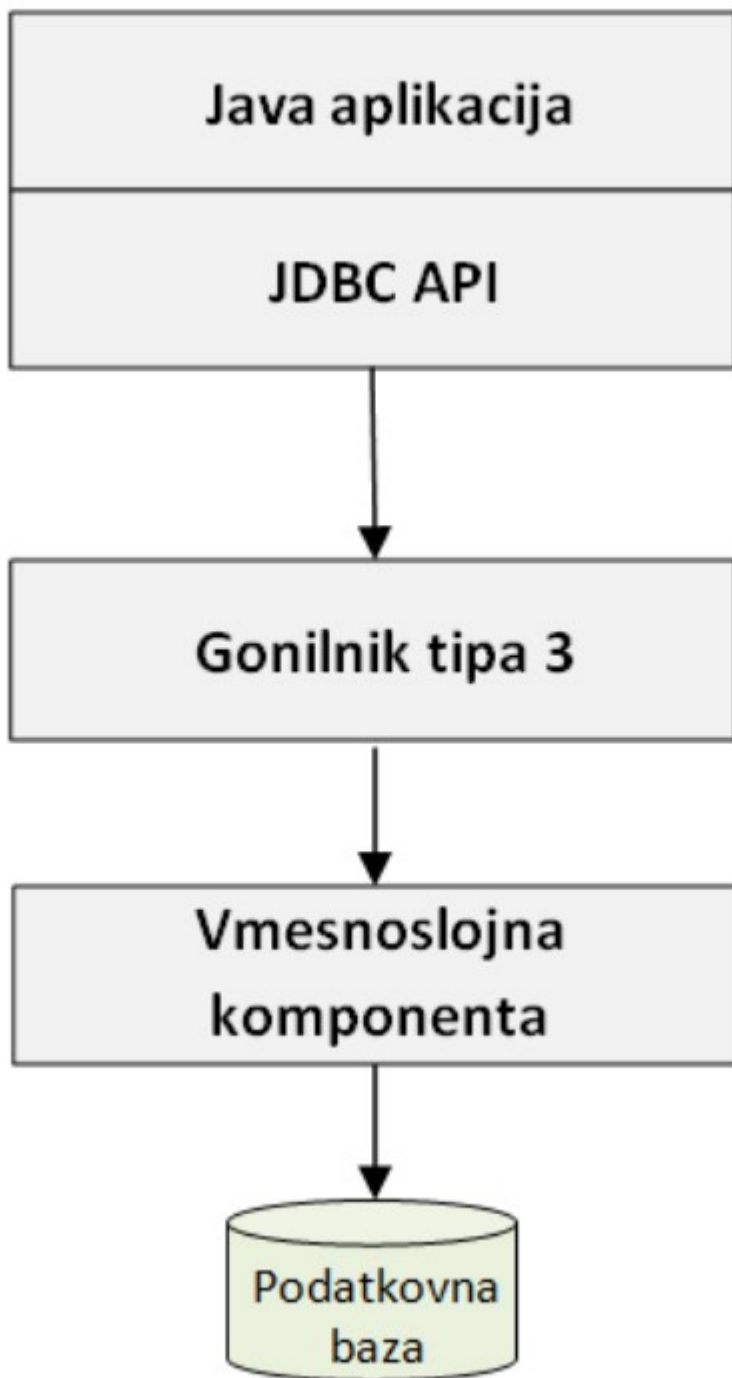
## Tip 2: delni javanski gonilnik

JDBC klic se posreduje specifičnemu gonilniku za posamezen tip podatkovne baze, vendar ni napisan v Javi, zaradi česar je v praksi redko uporabljen.



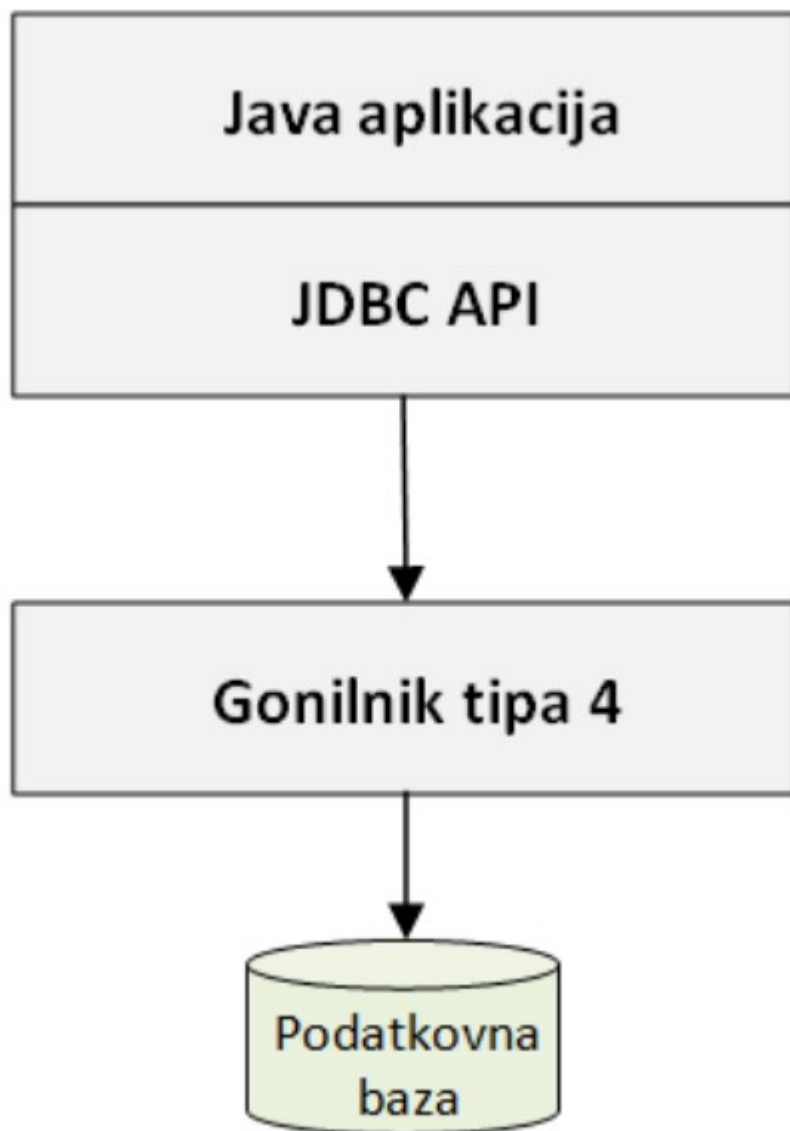
### Tip 3: javanski/mrežni gonilnik

Najbolj učinkovit gonilnik, vse zahteve se preko mreže posredujejo do vmesnega sloja, ki nato ustrezno pretvori ukaze (uporaba nekega drugega tipa gonilnika). Zahteva vzdrževanje gonilnikov na strani strežnika.



#### Tip 4: čisti javanski gonilnik

Omogoča komunikacijo direktno s podatkovno bazo, JAR datoteko dodamo v classpath. Omogoča visoko prenosljivost in neodvisnost ter dober performanse, vendar za vsak tip baze potrebujemo drug gonilnik.



Pošiljanje SQL stavkov podatkovni bazi:

- **Statement** - pošilja navaden SQL, ni dobra praksa, ker je navaden String
- **Prepared statement** - pošilja navaden SQL, predpošiljanje, lahko si PB pripravi način izvajanja, chekira za vdore
- **Callable statement** - PL/SQL (ni dobra praksa)

## Koraki pri uporabi JDBC

Korak 1: Nalaganje gonilnika

*Od Jave 6 ni več potreben, naloži se na podlagi JDBC URL niza.*

Korak 2: Sestavljanje URL niza za povezavo na bazo

**Format:** `jdbc:vendorName://host:port/databaseName`

```
String host = "jakmar.cloud.si";  
String dbName = "jakaStorage";
```

```
int port = 8080;
String db2Url = "jdbc:db2://" + host + ":" + port + "/" + dbName;
```

### Korak 3: Vzpostavljanje povezave

Povezavo pridobimo s pomočjo razreda `DriverManager` s klicem njegove metode `getConnection()`.

```
String userName = "jakmar17"
String password = "geslo123"
Connection con = DriverManager.getConnection(db2Url, username, password);
```

### Korak 4: Kreiranje objekta `Statement`, `PreparedStatement` ali `CallableStatement`

#### `Statement`

Omogoča izvedbo SQL stavka, ki ga sestavimo kot navaden String. Ne omogoča uporabe parametrov.

#### `PreparedStatement`

SQL povpraševanje sestavimo v obliki niza, ki omogoča uporabo parametrov. Parameter označimo kot `?`, ki ga nato ustavimo kot `setXXX(indeks, vrednost)` (npr `setInt(1, 3)`). Preprečuje SQL injection in je predhodno preveden - zagotavlja boljši performanse.

#### `CallableStatement`

Omogoča klic shranjenih procedur.

### Korak 5: Izvršitev SQL povpraševanj ali shranjenih procedur

#### Uporaba objekta `Statement`

```
public void vrniUporabnika (int id) {
    Statement s = null;
    try {
        s = conn.createStatement();
        String sql = "select * from uporabniki where id_uporabnika = " + id;
        ResultSet rs = s.executeQuery(sql);

        //obdelava rezultatov
        if (rs.next()) {
            String ime = rs.getString("ime");
            //nadaljna obdelava rezultatov
        } else {
            System.out.println("Ne najdem uporabnika");
        }
    } catch (SQLException e) {
        System.out.println(e.printStackTrace());
    } finally {
```

```

        if (s != null)
            s.close();
    }
}

```

Nad objektom tipa `Statement` vršimo eno izmed operacij:

### Operacija

<code>ResultSet executeQuery(String sql)</code>	vrne tabelo rezultatov tipa <code>ResultSet</code>
<code>boolean execute(String SQL)</code>	vrne <code>true</code> , če lahko pridobimo <code>ResultSet</code>
<code>int executeUpdate(String SQL)</code>	vrne število spremenjenih/dodanih/izbrisani vrstic

### Uporaba objekta `PreparedStatement`

```

public void vrniUporabnika(int id) {
    PreparedStatement ps = null;
    try {
        String sql = "select * from uporabniki where id_uporabnika = ?";
        ps = conn.prepareStatement(sql);
        ps.setInt(1, id);
        ResultSet rs = ps.executeQuery();

        //obdelava rezultatov
        if(rs.next()) {
            String ime = rs.getString("ime");

            //nadaljna obdelava
        } else {
            // uporabnika ne najde
        }
    } catch (SQLException e) {
        System.out.println(e.printStackTrace());
    } finally {
        if (ps != null)
            ps.close();
    }
}

```

### Korak 6: Obdelava rezultatov

Če ne poznamo strukture tabele, jo pridobimo s klicem metode `getMetaData()`.

```

ResultSet rs = ps.executeQuery();
while(rs.next()){
    String ime = rs.getString("ime");
    String priimek = rs.getString("priimek");
}

```

```
int starost = rs.getInt("starost");
System.out.println("ime: "+ime+" priimek: "+priimek+"
starost: "+starost);
}
```

Privzeto se lahko v `ResultSet` premikamo samo naprej, lahko pa definiramo scrollable `ResultSet`:

```
Statement s = con.createStatement (
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = s.executeQuery("select * from table");
```

## Korak 7: Zapiranje povezave

```
try {
    if (conn != null)
        conn.close();
} catch(SQLException e) {
    e.printStackTrace();
}
```

*"Imate prijaznega asistenta letos.... šalim se malo"*

Jurič, 21.10.19

## JDBC transakcije

Privzeto se vsi ukazi samodejno potrdijo, torej `autoCommit = true`

Če želimo več ukazov izvesti kot eno transakcijo:

```
Connection conn = DriverManager.getConnection(url, user, pass);
connection.setAutoCommit(false);

try {
    statement.executeUpdate(...);
    statement.executeUpdate(...);
    statement.executeUpdate(...);
    ...

    conn.commit();
} catch (SQLException e) {

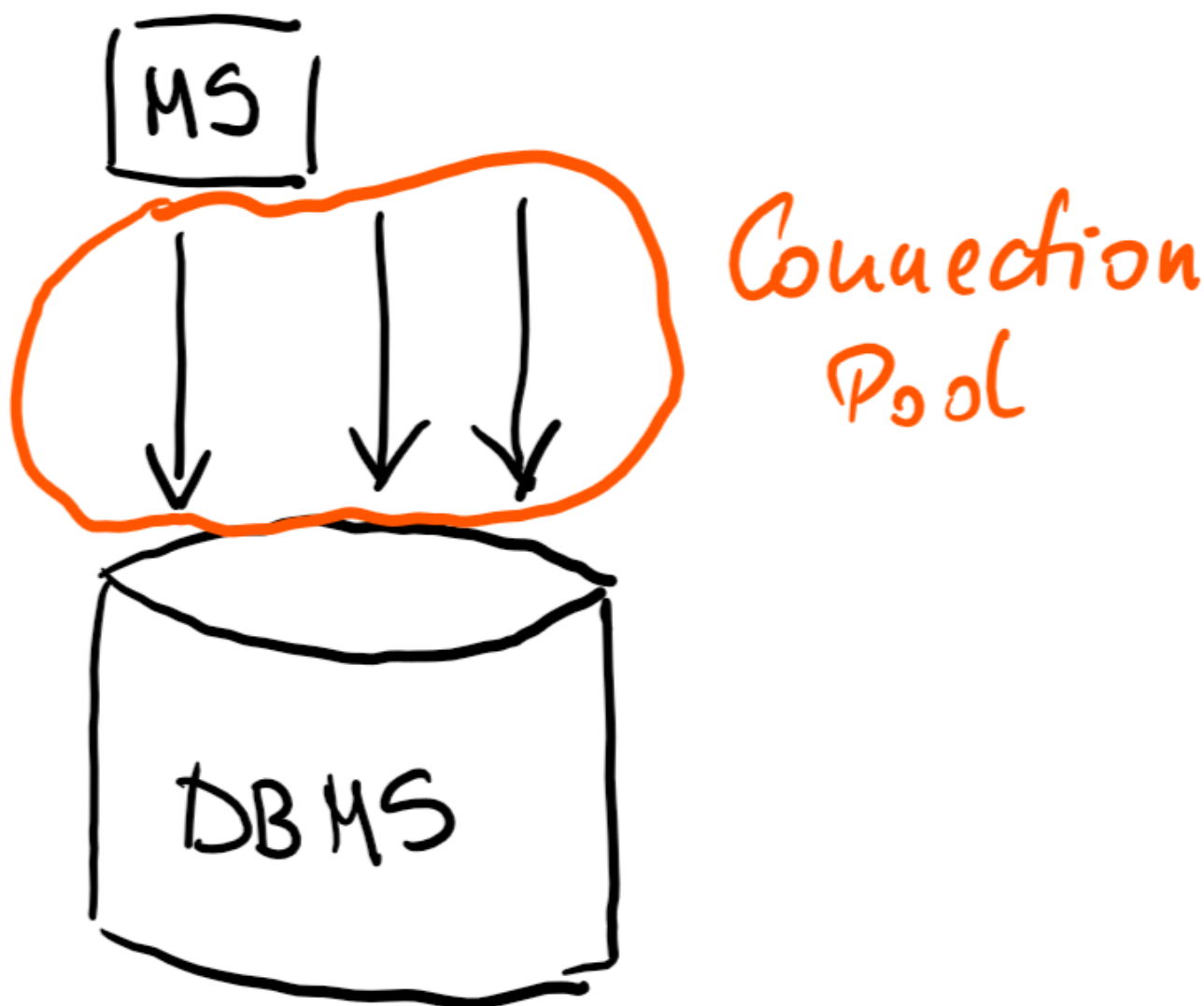
} finally {
    try {
        if (conn != null)
            conn.close();
    }
}
```

```
    } catch (Exception e) {  
  
    }  
}
```

## JDBC Connection Pool

Možnosti povezave na podatkovno bazo:

1. Vsaka mikrostoritev ima svojo povezavo (slaba izkoriščenost povezav)
2. Koncept bazen povezav - določeno število vnaprej pripravljenih povezav na PB. Vsaka storitev se poveže na connection pool. Storitve ima občutek, da ima svojo povezavo, v resnici bazen sproti določa povezavo, glede na potrebo storitve.



*Connection Pool* == **DataSource** kje se nahaja programerja ne zanima (mikro storitev, PB...) **JNDI** = Java Naming and Directory Interface, abstrakcija LDAP-ja

Primer uporabe JNDI-ja



```
public Connection povezi() throws SQLException {
    Connection con = null;

    try{
        Context initCtx = new InitialContext();
        Context envCtx = (Context) initCtx.lookup("java:comp/env");

        DataSource ds = (DataSource)envCtx.lookup("jdbc/TestDB");
        con = ds.getConnection();
    } catch (NamingException e) {

    }

    return con;
}
```

"Pasvord"

Jurič

"Kolegice in kolegi"

Jurič

## RDBS TPM

## Dobre prakse uporabe JDBC

- Ne-mešanje poslovne logike in JDBC

Ne delaj tega

```
public boolean preveriStanjeUp (...) {
    //JDBC koda
    ...
    //

    //poslovna logika
    ...
    //
}
```

Delaj tako:

```
public boolean preveriStanjeUp (...) {
    //posebej narediš DAO (DataAccessObject)
    getStatus(...);

    //naprej pišemo poslovno logiko
```

```
...
}
```

- uporaba transakcij
- uporaba `PreparedStatement` ne `Statement`

## DAO (Data Access Object)

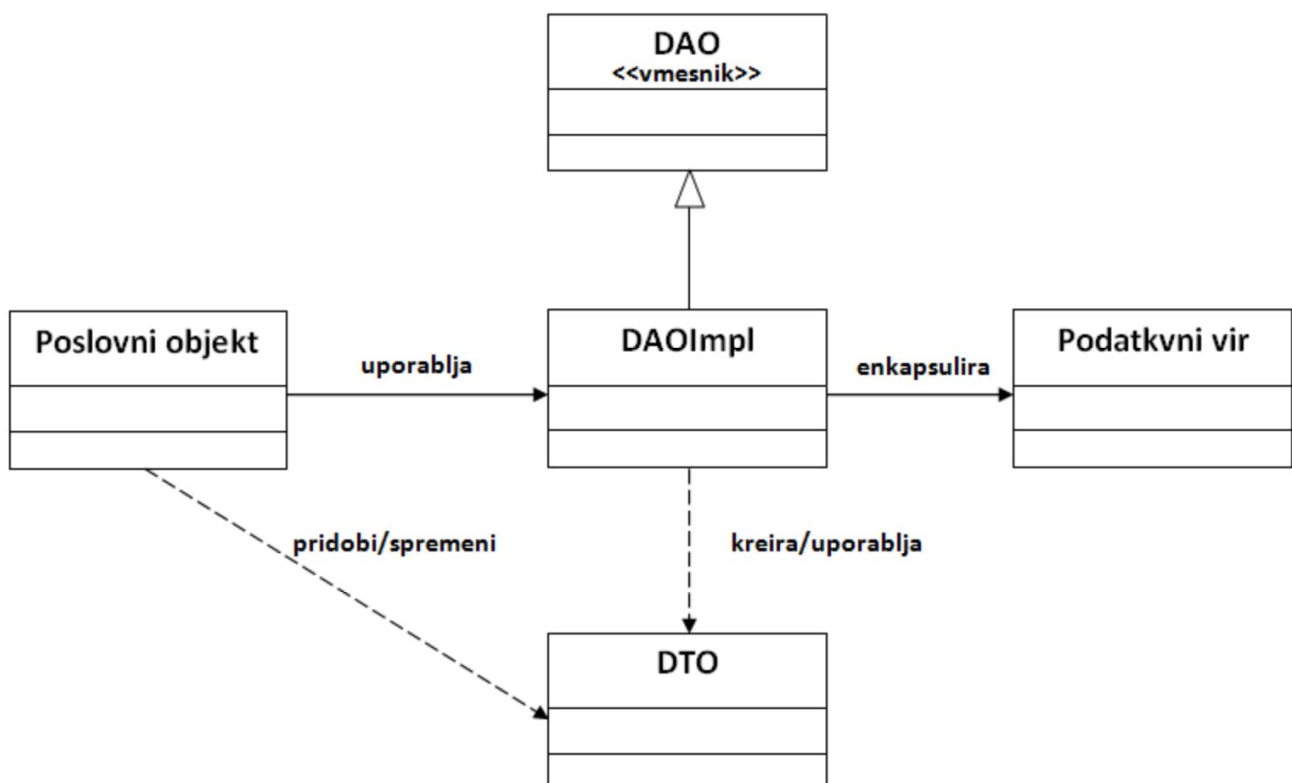
Standardni javanski načrtovalski vzorec, ki predvideva ločitev nizko-nivojskih operacij za dostop do podatkov od visoko-nivojske poslovne logike. DAO skriva kompleksnost in izpostavlja vmesnik, ki ga uporablja poslovni nivo. Dao se uporablja v kombinaciji z **objekti za prenos podatkov DTO** (Data Transfer Object).

V DAO imamo tudi `getUporabnik()`, kjer dobimo "pravega uporabnika". Za ta namen uporabimo `Java Zrno` (`Java Bean`) s prilagojenimi `get` in `set` metodami. Takšnemu zrnu pravimo `DTO` (`Data Transfer Object`).

Zakaj?

- razdelimo odgovornost
- lahko spreminjamo podatkovno bazo brez spreminjanja poslovne logike

Vzorec DAO



Naloge DAO

- transakcije
- obravnava napak
- beleženje (logiranje)

## Generiranje baznega DAO

```
public interface BaseDao {  
  
}
```

## Serializacija

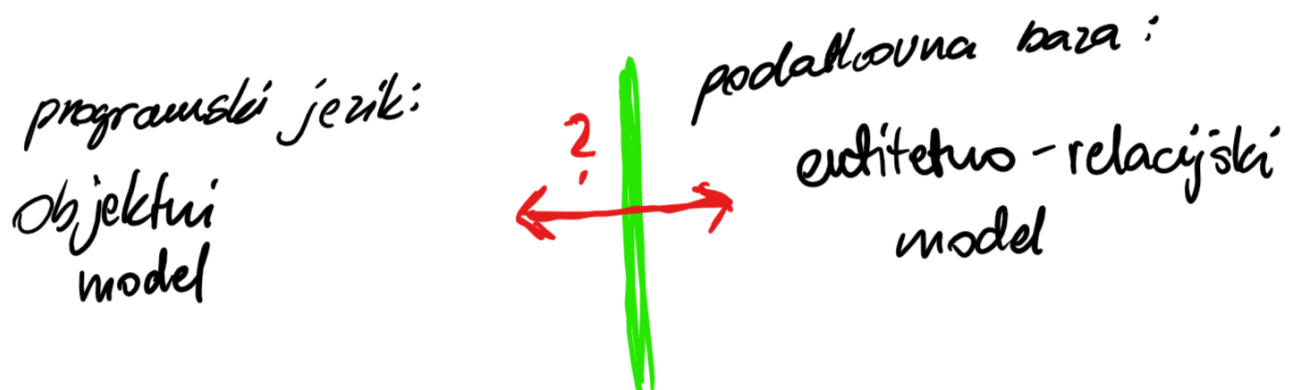
V eni JVM se objekti pošiljajo kot *pass-by-reference*. Avtomatski postopek pretvarjanja iz stanja objekta v tok podatkov za pošiljanje objekta med različnimi JVM v omrežju. Obstajata dve vrsti serializacije:

- binarna serializacija (podpira npr. ciklične grafe)
- markup serializacija (JSON ali ??) - pretvarjanje direktno iz in v objekte (podpira zgolj hierarhične podatkovne modele)

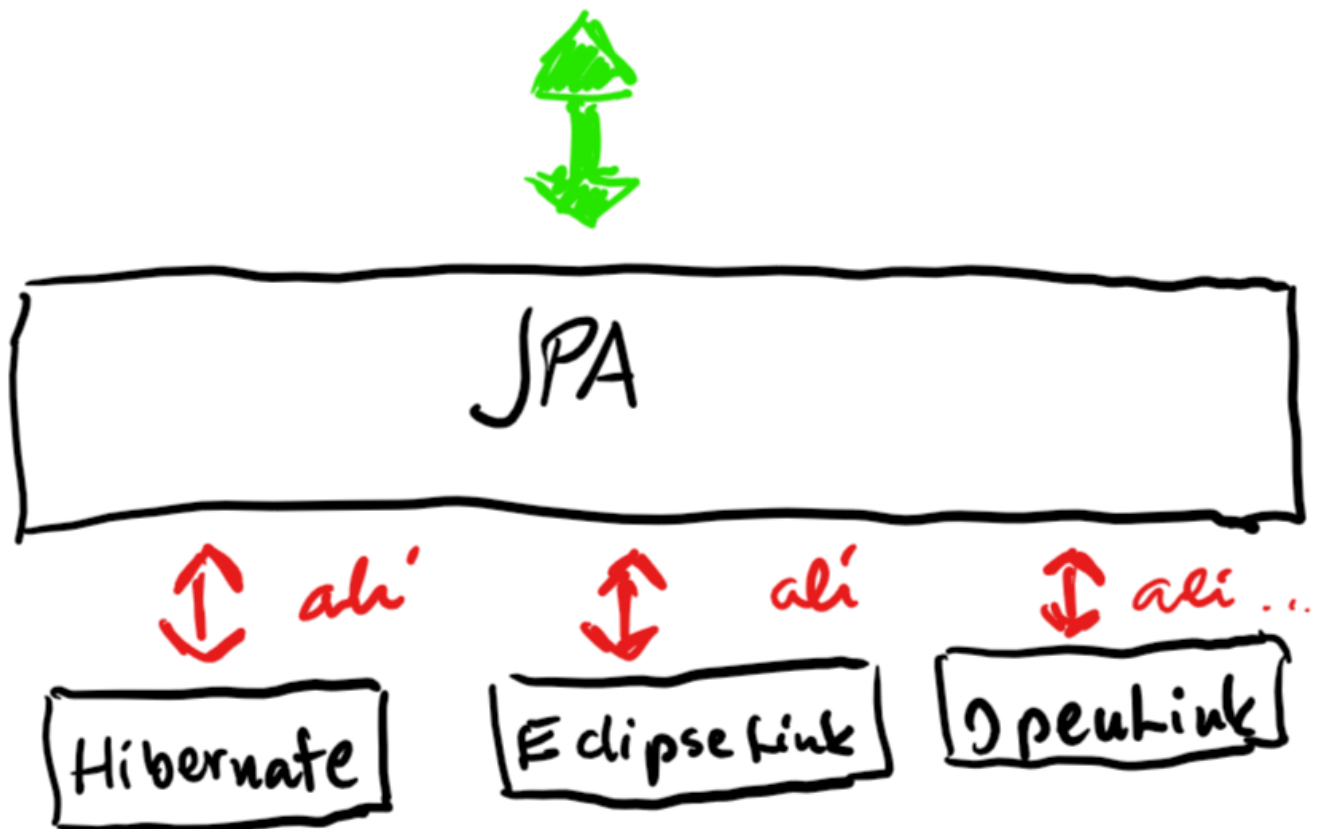
```
public Razred serializabilen implements Serializable {  
  
}
```

## Java Persistence API (JPA)

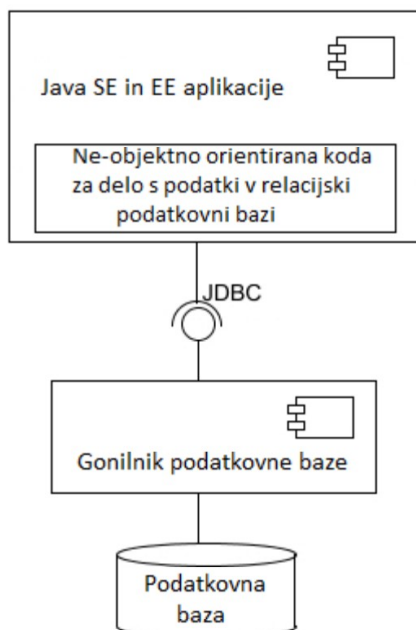
Namesto, da za vsak objekt napišemo DAO in DTO, ali lahko napišemo samo DTO?



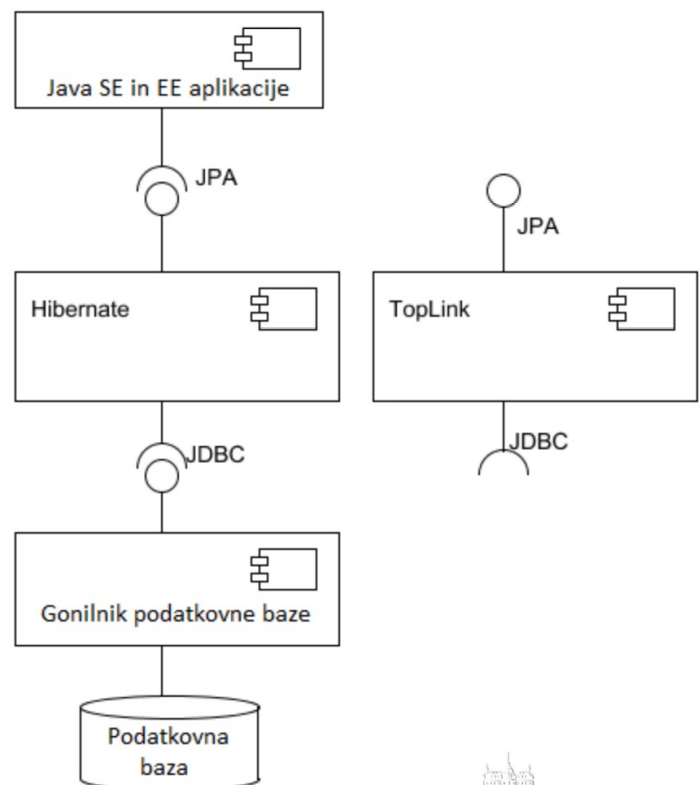
Za to poskrbi **Objektno-relacijski preslikovalniki (ORM)**, ki obstajajo v večini programskih jeziki. Najbolj poznano je *Hibernate*, v Javi se programski vmesnik imenuje **JPA - Java Persistence Application**.



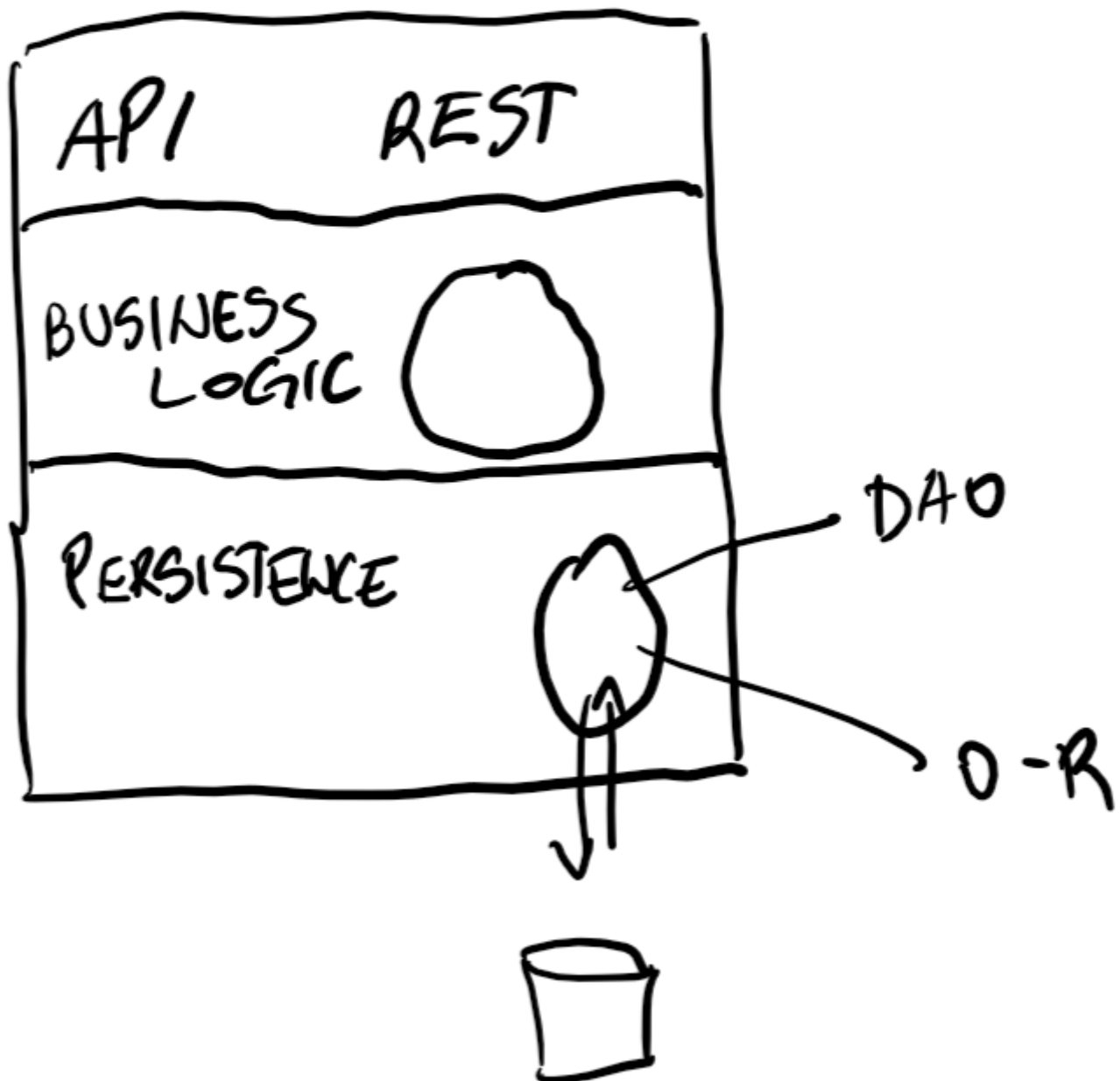
## Standardni JDBC



## Java Persistence API



UI



## Anotiranje entitetnih razredov

@Table spreminja privzeto ime tabele (drugače uporabimo ime razreda)

@Column spreminja privzeto ime stolpca (drugače uporabimo ime spremenljivke)

```
//koda iz slajda 28
```

## @SecondaryTable

```
@Entity
@SecondaryTables
```

```
//slajd 29
```

## @Id

Označuje atribut, ki definira primarni ključ

## @Temporal "Kalendar"

## @Transient

Spremenljivka, ki je nočemo, da je shranjena v podatkovni bazi.

## @Enumerated

## @ElementCollection in @CollectionTable

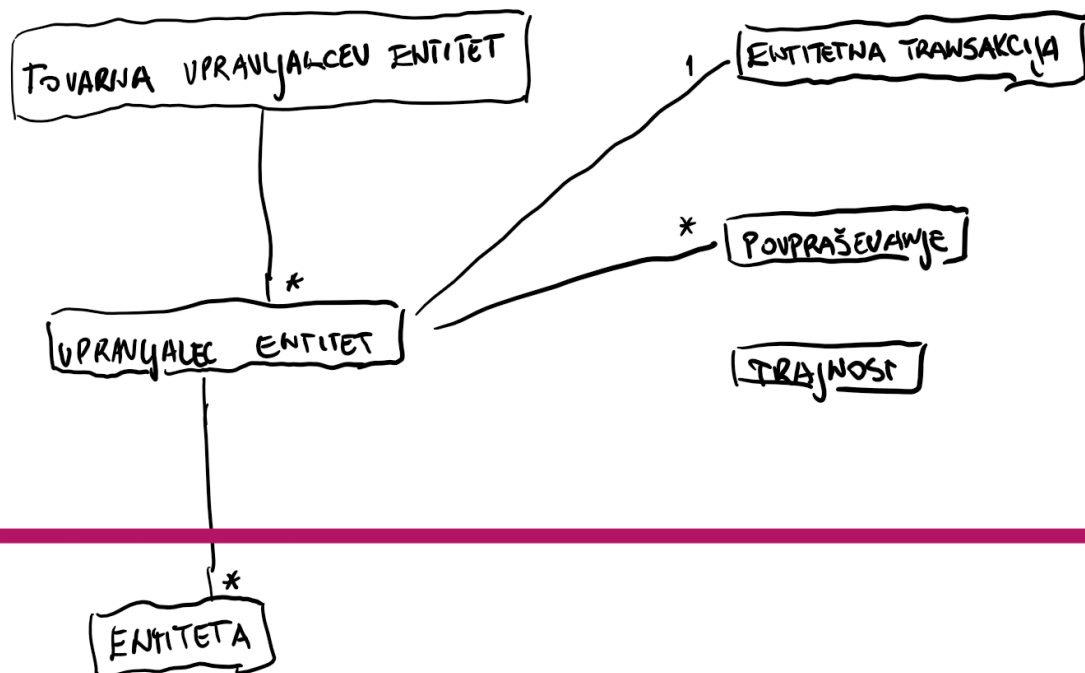
```
@Entity
public class Oseba {
    @Id
    private String id;
    private String ime;
    private String priimek;
    @ElementCollention (fetch = FetchType.LAZY) //LAZY ali EAGER
    @CollectionTable (name = "Zaznamek")
    @Column (name = "vrednost")
    private ArrayList <String> zaznamki;
}
```

## Relacije med entitetami

## Dedovanje

## Struktura Java Persistence APIja

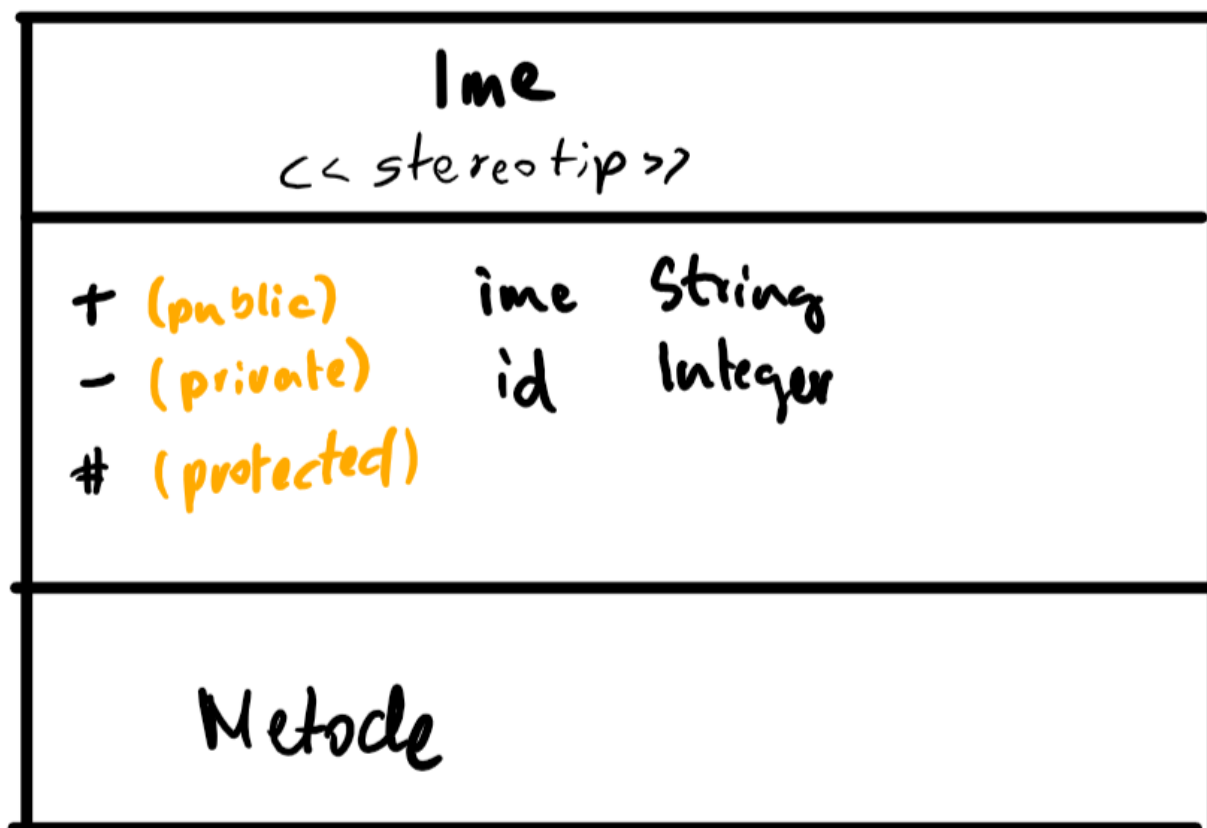
javax.persistence



"Kolegice in kolegi, tako mimogrede" **Design patterns** Najboljše rešitve nekih tipičnih problemov s katerimi se srečamo pri programiranju.

Zbirka design paternov **GoF** - Gang of Four. [Mogoče ta](#), [verjetno ta](#).

## UML CLASS DIAGRAM



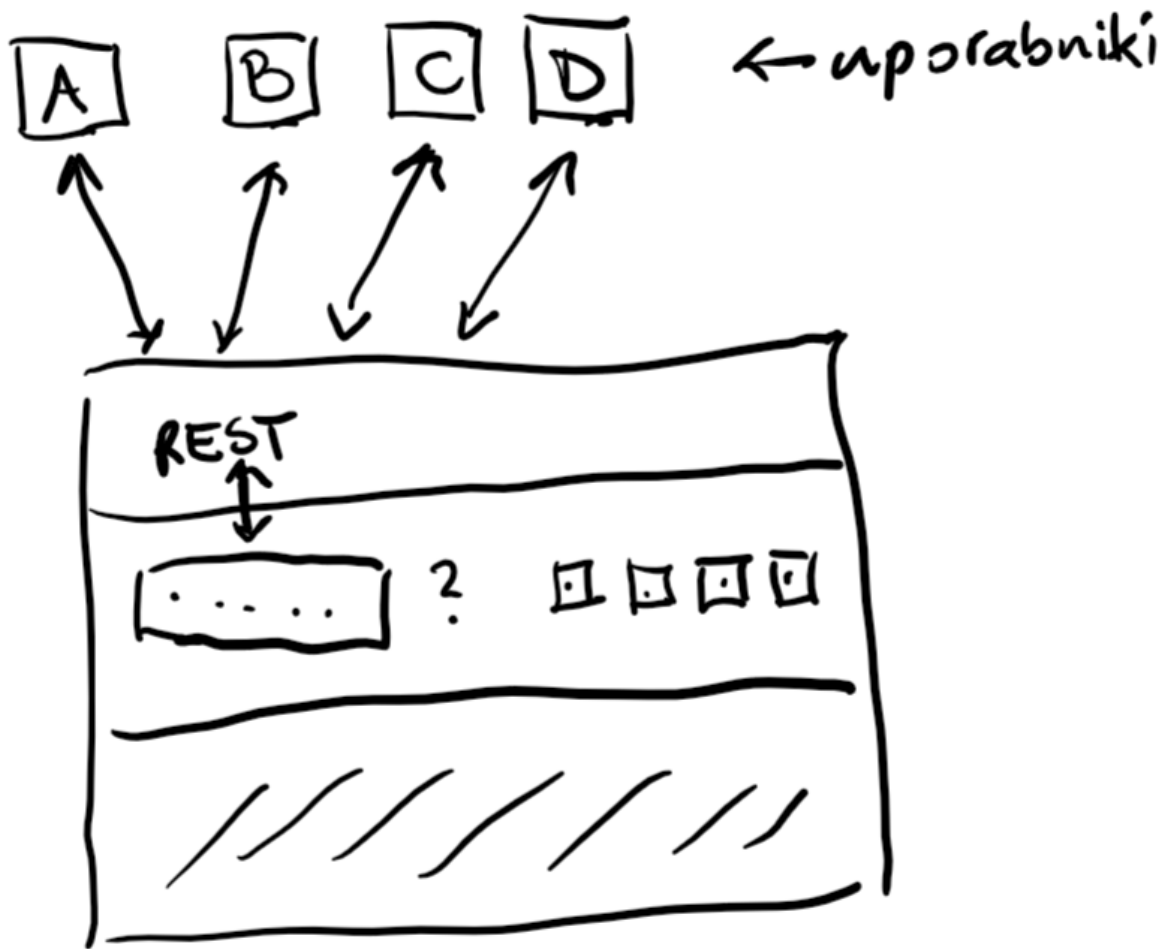
spremenljivke in metode lahko izpustimo

## Nivo poslovne logike in CDI

### CDI - Contexts and Dependency Injection

1. Zagotavlja kontekst izvajanja - komponente imajo določen življenski cikel in iteracije glede na jasno definirane in razširljive kontekste





2. Vstavljanje odvisnosti omogoča vstavljanje referenc na posamezne komponente znotraj aplikacije

CDI zrna

So razredi, ki jih instancira, upravlja in vstavlja CDI vsebnik.

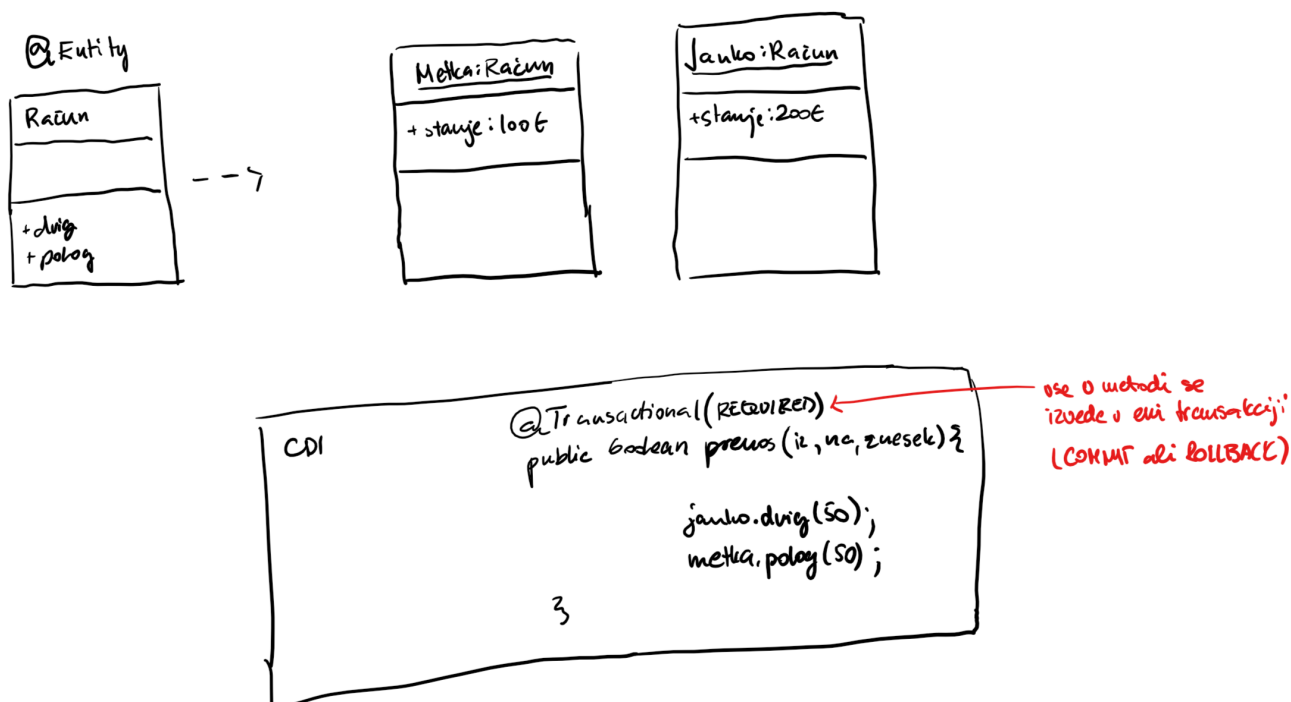
## CDI container

CDI vsebnik skrbi za:

- življenski cikel
  - vsebnik sam ustvarja nove instance razreda (določamo samo scope - `request`, `session`, `application`)

Scope	Annotation	Description
Request	@RequestScoped	Single HTTP request
Session	@SessionScoped	Multiple HTTP requests
Application	@ApplicationScoped	Shared state across all interactions in a web application
Conversation	@ConversationScoped	multiple invocations of the JavaServer Faces lifecycle
Dependent	@Dependent	Its lifecycle depends on the client it serves (Default scope)
Singleton	@Singleton	State shared among all clients

- vstavljanje odvisnosti



"Kaj bluzi Jurič"

Jurič, 4.11.2019

## Interceptor

Metoda, ki omogoča, da se pred ali po izvedbi neke metode izvedemo še neko drugo kodo (metodo) avtomatsko.

```
@MojPrestreznik
public void nekaDrugaMetoda (int id) {
    //preden se izvede nekaDrugaMetoda se izvede metoda MojPrestreznik
}
```

`MojPrestreznik()` je definiran v CDI znu.

## CRUD ukazi

**C** - create, **R** - read, **U** - update, **D** - delete ukazi

## Loggiranje v Javi

---

### Knjižnice, ki podpirajo loggiranje

- JUL (Java Util Logger) - že vključena
- LOG4J 1 in 2
- SLF4J

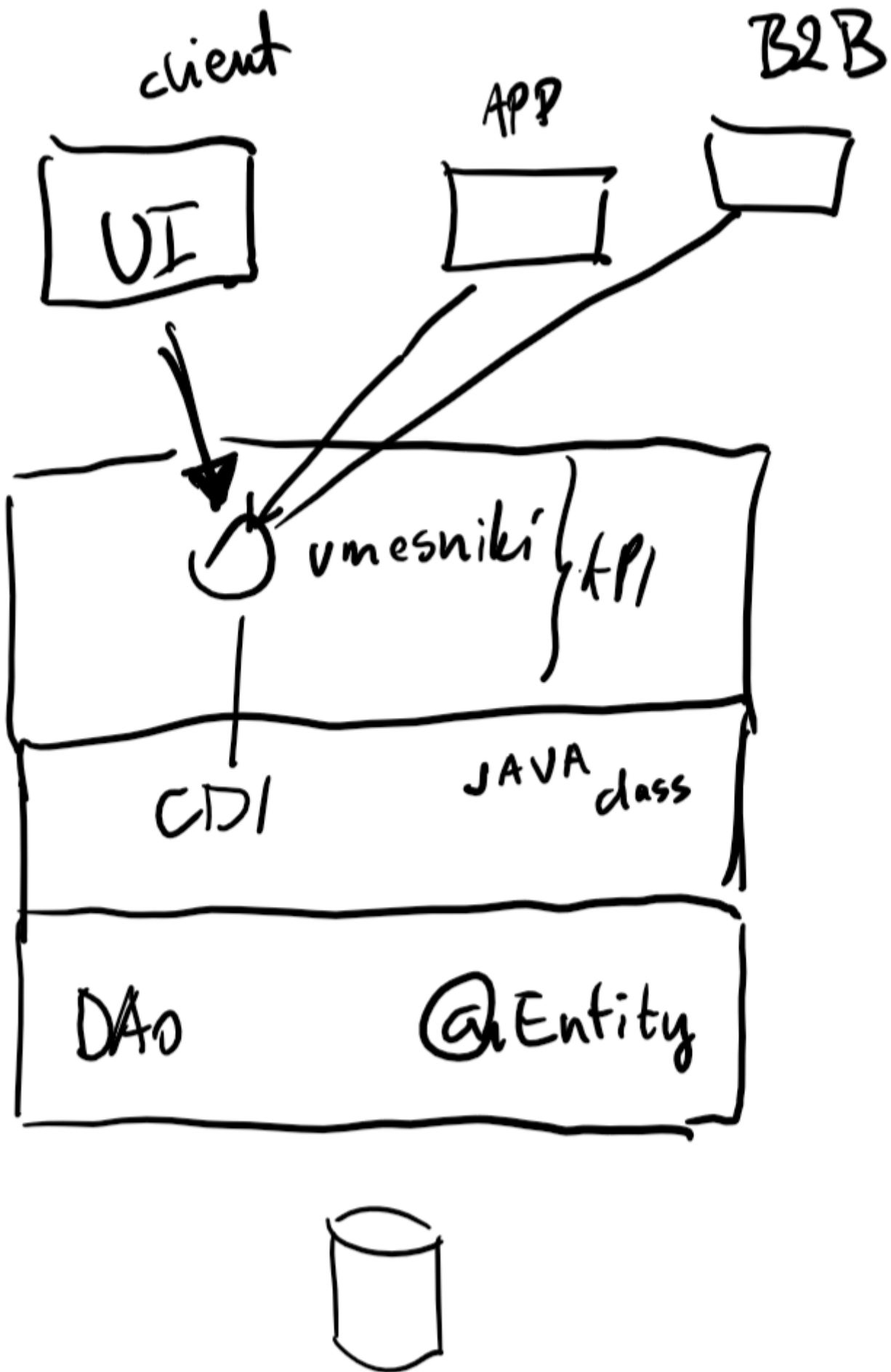
*Pol so prišli malo boljši časi, manj stari časi. Jurišič, 11.11*

## REST

---

### Zgodovinski razvoj vmesnikov

- binarni
  - RPC
  - CORBA
  - RMI
- text
  - SOAP (XML), opišemo z WSDL
  - REST (JSON, XML, ali katerikoli MIME), opis ni potreben (ponavadi Swegen/Open API)
- gRPC (spet binarni)



## API in HTTP metode

```
vrniSeznamArtiklov();
izvediPlacilo();
dodajArtikel();
posodobiArtikel();
```

Za dostop do metod uporabimo **HTTP** metode:

HTTP metoda	URL	
GET	/razmerje	pridobi seznam razmerij
GET	/razmerje/345	pridobi razmerje z <i>DI</i> 345
POST	/razmerje	ustvari novo razmerje
PUT	/razmerje/345	posodobi razmerje z <i>ID</i> 345
DELETE	/razmerje/345	izbriše razmerje z <i>ID</i> 345

Vire oblikujemo enostavno in učinkovito (grobo zrnato, samostalniki v množini) Ustvarjamo lahko nove pod vire  
GET `artikel/345/akcija`

Vir zbirke	Vir instance
/razmerja	/razmerja/id_razmerja

"A se še kaj spomnite slovenščine iz srednje šole? Samostalnik, pridevnik,... Šalim se, saj vem da se" Jurič

GET za branje vira

GET za branje določenega vira

GET na viru

Tip	URL
Ostranjevanje	/artikli?start=0;offset=0
Filtracija (iskanje)	artikli?q='...', artikli?where=vrednost:gte:512
Sort	artikli?sort='...', artikli?order=naziv ASC, prioriteta DESC

POST za ustvarjanje

``PUT`` za posodabljanje

DELETE za brisanje

Medklic: minor/major verzije in kompatibilnost za nazaj

**Minor** verzije (1.0, 1.1, 1.2) so kompatibilne za nazaj **Major** verzije (1.x, 2.x) niso kompatibilne za nazaj - različne major verzije imajo svoj url `api.url/v1/...`, `api.url/v2/...`

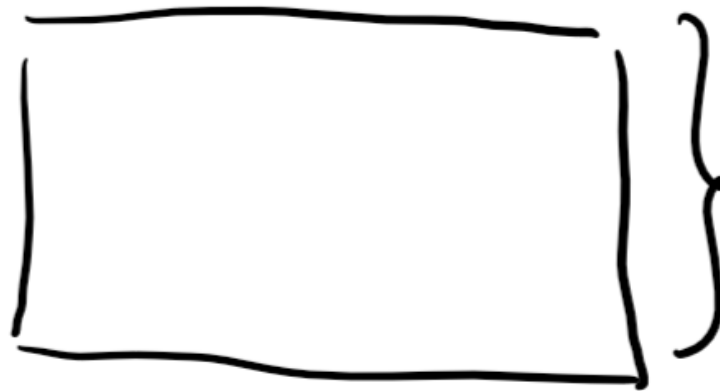
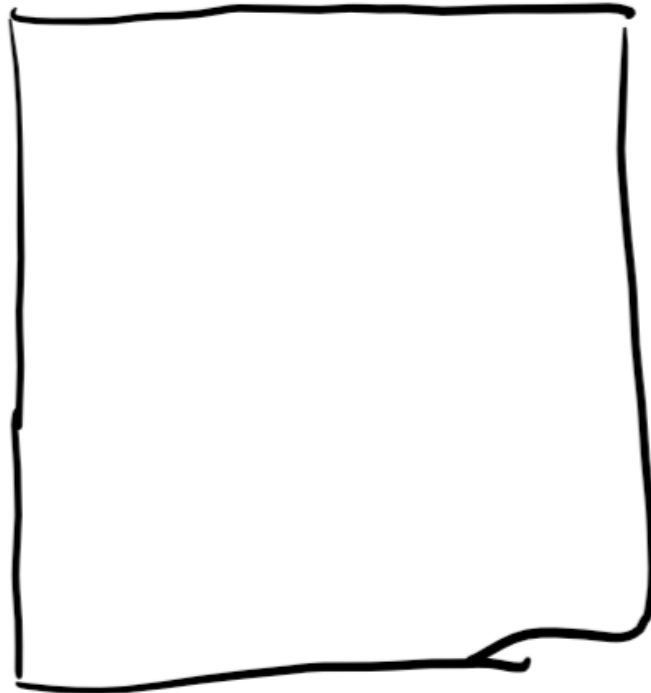
## Fornat sporočil

### JSON

#### JSON header

Header si zamislimo sami (pri vseh API klicih naj bo *bolj ali manj* enak). V njem definiramo podatke kot so vsi artikli, preneseni artikli, *offset*...

{

} JSON  
header} body  
(artikli)

}