

TDP024

(Program to an Interface) , Designmönster,Datalager, Logiklager

Anders Fröberg anders.froberg@liu.se

Program to an interface

- **"Program to an interface"** är en metod att organisera och bygga sitt system på som är grundläggande för alla system som skall vara modulära och hållbara.
- För en programmerare är det bland de viktigaste kunskaper man kan ta med sig in i ett projekt.
- Projekt som väljer enklare metoder, eller hittar genvägar runt, har en drastiskt förkortad livslängd.
- Det är på abstraktionsnivån *programmering* som denna metod befinner sig, inte på samma abstraktionsnivå som SCRUM eller arkitektur, etc.

Dependency Injection

```
public static void main(String [] args) {  
    ListUtil listUtil = new ListUtil(new QuickSort());  
    int min = listUtil.findSmallest([21,13,55,34])  
}
```

Någon gång måste man välja implementation, i detta fall är det koden som skapar ett nytt objekt av ListUtil som måste välja vilken implementation som skall användas.

För den intresserade så går det även att få bort detta beroende. Med hjälp av *"Inversion of Control"* kan man få "program to an interface on steroids".

Du anger aldrig vilken implementation som skall användas i koden, utan detta löses "runtime" från t ex en XML fil som säger vilken implementation av Sort som skall användas just nu.

Vi har inte tid med "Inversion of Control" i denna kurs, men det rekommenderas till den intresserade.

Motivation & Problem

- Sparande och förmedling av information är det centrala i våra tjänster
- Databaser gör det möjligt för oss att spara information på ett säkert och stabilt sätt
- Data bör komma in och ut ur applikationen på ett naturligt sätt
- En SQL databas är en samling tabeller med rader och kolumner
- En Java applikation består utav objekt
- JPA är ett Java bibliotek (del utav Java EE) som "översätter" Java klasser till rader och kolumner - ORM (Objekt-Relational-Mapping)



JPA – POJO & @Entity

- Instanser av klasser annoterade med @Entity kan sparas till databasen
- Som ett minimum behöver man annotera ett fält med @Id för primary-key
- Egentligen är klassen en POJO (Pure Old Java Objekt) utan några speciella krafter
- @Column används för att konfigurera kolumnen för ett fält (nullable, unique, etc)
- JPA kommer skapa tabeller som anses nödvändiga
- Kan hantera komplexa relationer mellan klasser och även datatyper så som HashMap och List (med lite konfiguration)

JPA - Entity Exempel

Interface

```
public interface Todo {  
  
    long getId();  
  
    void setId(long id);  
  
    String getTitle();  
  
    void setTitle(String title);  
}
```

Implementation

```
@Entity  
public class TodoDB implements Todo {  
  
    @Id  
    private long id;  
    private String title;  
    private String body;  
  
    @Override  
    public long getId() {  
        return id;  
    }  
    ...  
}
```

JPA – Table Example

id - bigint(20)	title - varchar(255)	content - varchar(255)
1	Title 1	Content 1
2	Title 2	Content 2
3	Title 3	Content 3
...	...	...
...	...	...



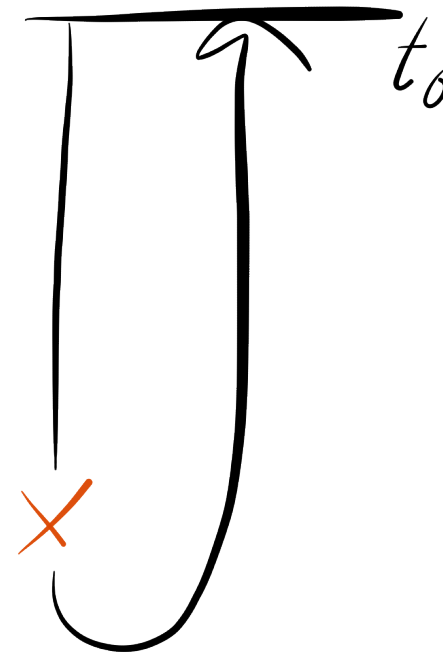
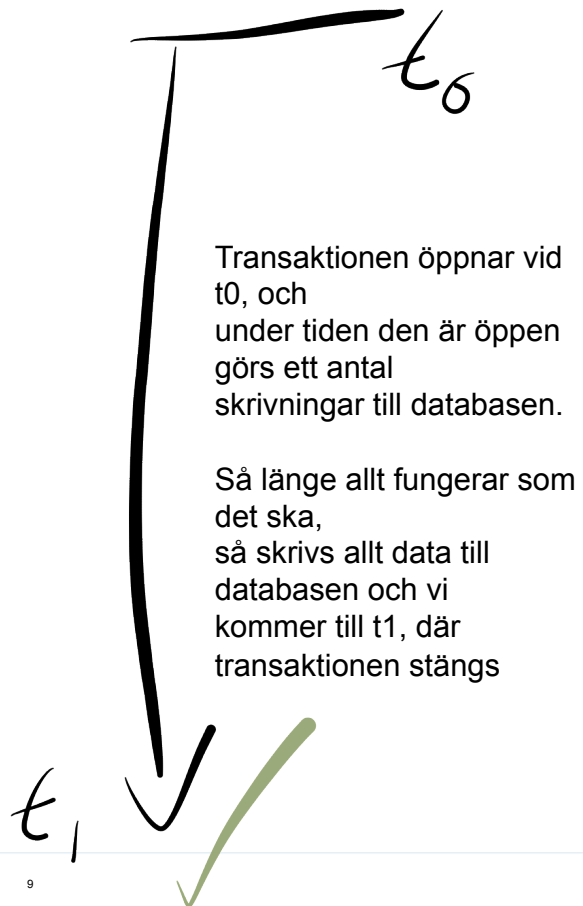
Behöver inte fundera på vilka tabeller eller kolumner som bör skapas, detta görs åt mig.

Innebär att det är lättare att byta databas den dagen företaget väljer att köpa in t ex Oracle, eller helt byta till t ex MongoDB (som inte består utav rader och kolumner alls)

EntityManager & Transactions

- Förutom att använda *annotations* (@) på klasser som skall sparas i databasen så behöver vi **kommunicera** med databasen.
- En **EntityManager** skapas genom klassen **EntityManagerFactory**
- EntityManager ansvarar för att skapa, starta och avsluta **transactions**
- All kommunikation som skriver till databasen måste ske inom en transaktion
- Antingen så lyckas alla operationer inom en transaktion eller så misslyckas alla

Transactions



Om något går fel inom transaktionen, så görs först en "rollback" på alla operationer, och transaktionen stängs. Allt är som det var vid t_0 .

Glöm inte bort EntityManager,
EntityManagerFactory och Transactions, vi
återkommer till dessa strax!

Ett **design pattern** är ett namn på en "konstruktion" eller uppsättning klasser som jobbar tillsammans på ett sätt som är användbart för många applikationer.

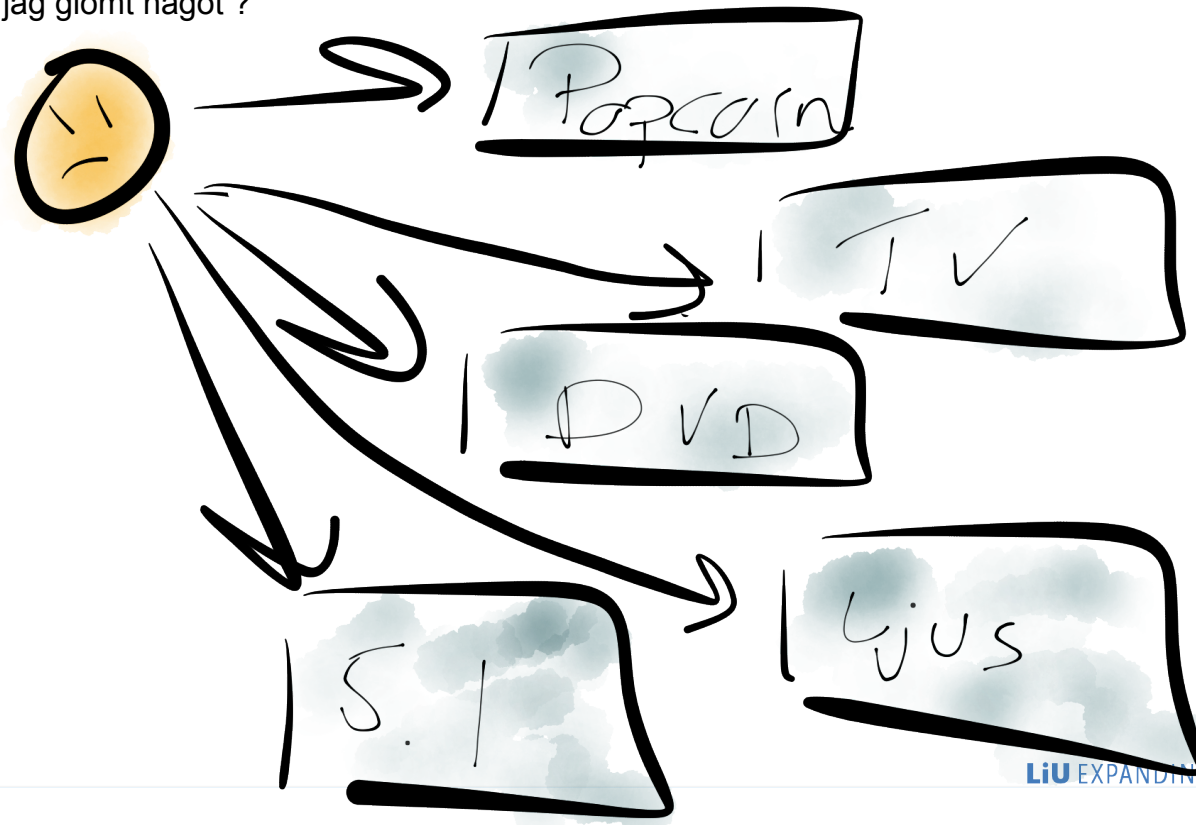
Det ger utvecklare ett vokabulär att kommunicera med istället för att upprepa sig själva hela tiden.

Intermezzo

FAÇADE DESIGN PATTERN

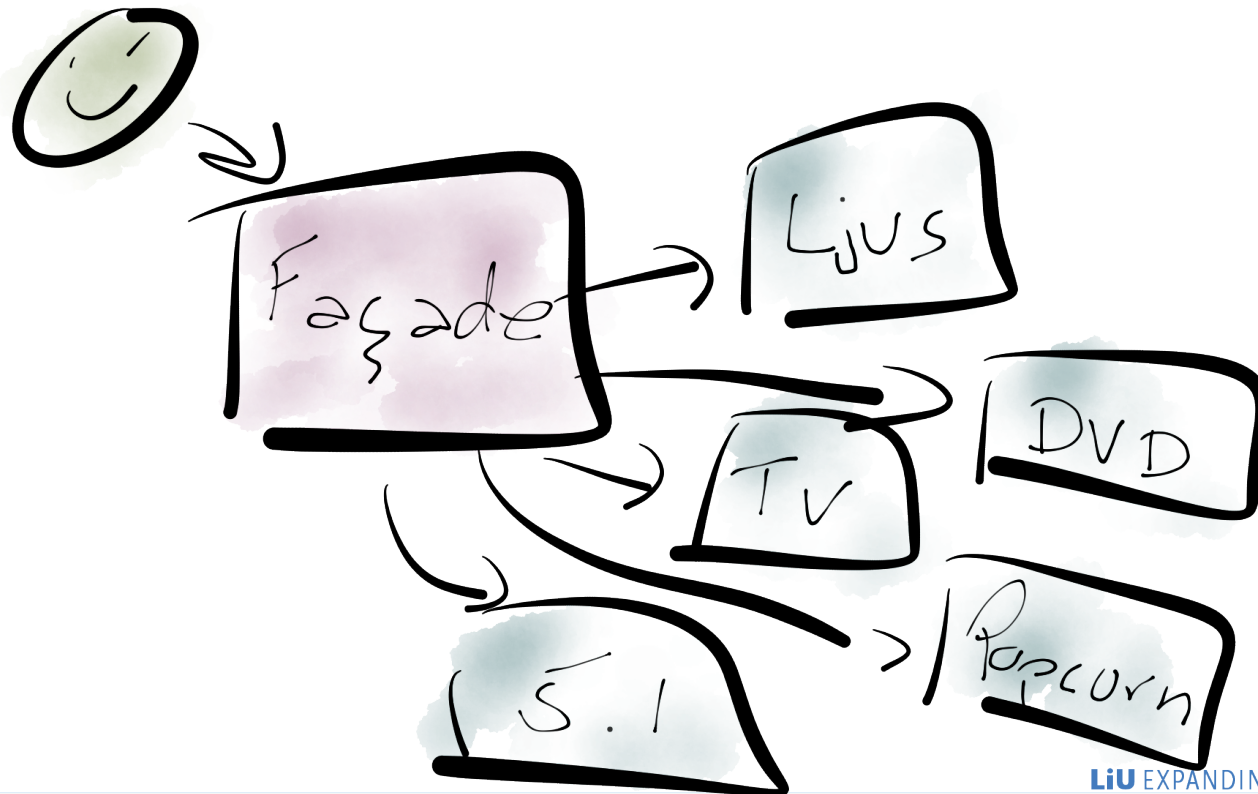
Intermezzo – Façade Design Pattern

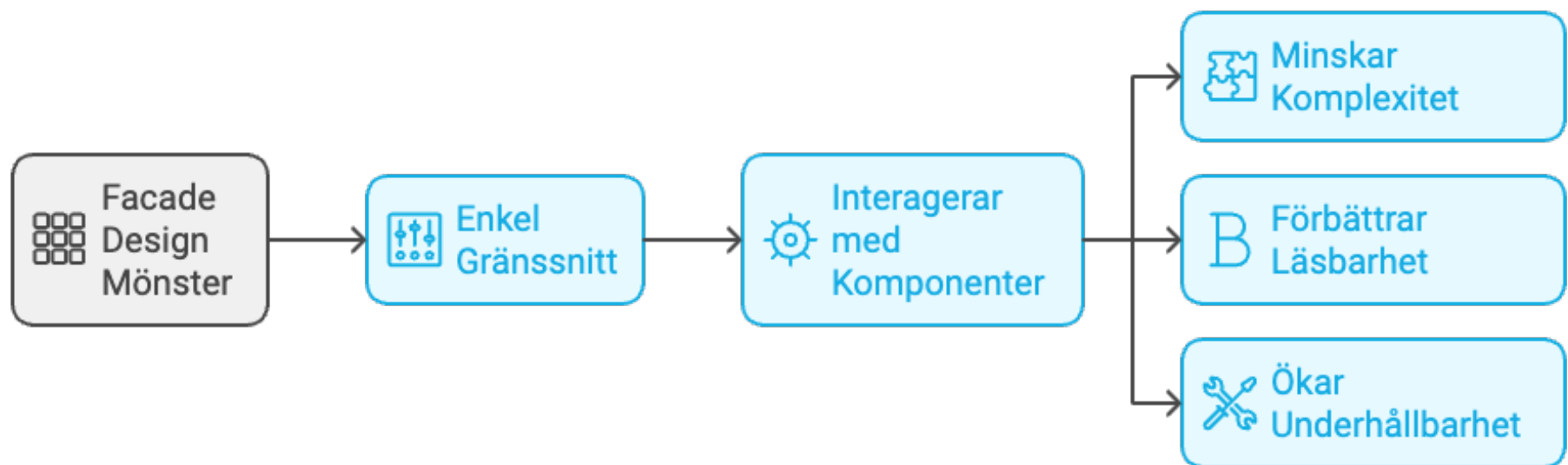
Har jag glömt något ?

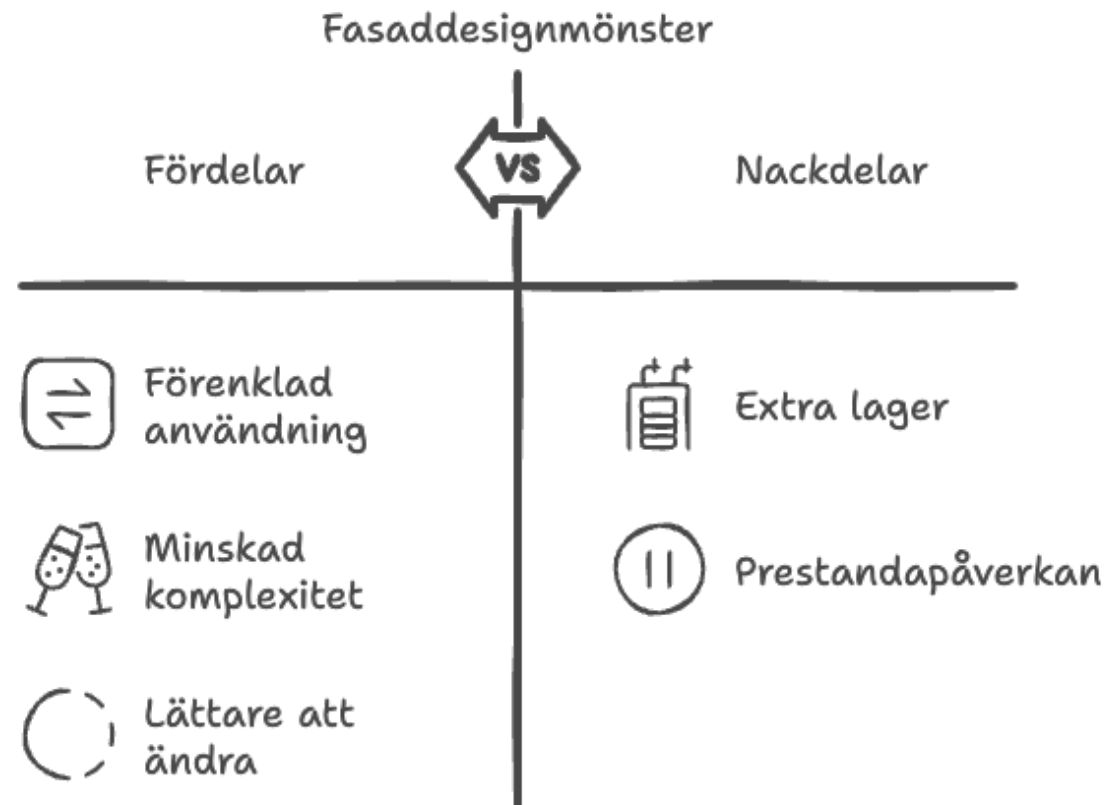


Intermezzo – Façade Design Pattern

Klart!







```
// Komponenter
class Engine {
    public void start() {
        System.out.println("Motorn startar...");
    }
}

class Tires {
    public void inflate() {
        System.out.println("Däcken pumpas...");
    }
}

class Lights {
    public void turnOn() {
        System.out.println("Ljusen tänds...");
    }
}
```

```
// Facade
class CarFacade {
    private Engine engine;
    private Tires tires;
    private Lights lights;

    public CarFacade() {
        this.engine = new Engine();
        this.tires = new Tires();
        this.lights = new Lights();
    }

    public void startCar() {
        lights.turnOn();
        tires.inflate();
        engine.start();
        System.out.println("Bilen är redo att köra!");
    }
}
```

Användning av Facade

```
public class Main {
    public static void main(String[] args) {
        CarFacade carFacade = new CarFacade();
        carFacade.startCar();
    }
}
```

Intermezzo - Facade Design Pattern

- En klass vars uppgift det är att abstrahera komplexa delar, och på så vis förenkla interaktionen med systemet
- I **datalagret** använder vi façades för att erbjuda funktioner för "verben" ***create, remove, update, find***
- Det är viktigt att tänka på att de façades som vi skapar i datalagret skall hantera **generiska** "verb" som är lika mellan applikationer, inte saker som är unika för just din tjänst.
- Vi har även façades i **logiklagret** som kan hantera andra typer av "verb", så som "checkIn" och "checkOut" ... mer om detta på senare föreläsningar

JPA – Façade, EMF, Transactions

```
public class TodoEntityFacadeDB implements TodoEntityFacade {
```

```
    @Override
```

```
    public long create(String title, String body) {
```

```
        EntityManager em = EMF.createEntityManager();
```

```
        em.getTransaction().begin();
```

```
        Todo todo = new TodoDB();
```

```
        todo.setTitle(title);
```

```
        todo.setBody(body);
```

```
        em.persist(todo);
```

```
        em.getTransaction().commit();
```

```
        return todo.getId();
```

```
    }
```

```
}
```

EMF är en klass som man måste skapa själv, det finns en sådan i kodskelettet .

Exemplet är inte komplett då vi inte hanterar eventuella fel på ett bra sätt. Mer om detta senare.

Interface på vänstersidan, implementation på höger!

LiU EXPANDING REALITY

JPA - Konfiguration

- JPA behöver **konfigureras** för att veta vilka klasser som skall sparas
- JPA behöver information om var databasen är och hur vi kopplar oss mot den
- `src/main/resources/persistence.xml`
- Vi kommer använda en databas som bara existerar i RAM, så inga ändringar behövs för detta.
- Finns en hel del andra inställningar som är viktiga, t ex "drop-and-create"

En grundläggande persistence.xml finns i kodskelettet.

persistence.xml

Lägg till en sådan rad för varje entity

...

```
<persistence-unit name="todo-dataPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>tdp024.todo.data.db.entity.TODO</class>
  <shared-cache-mode>NONE</shared-cache-mode>
  <validation-mode>NONE</validation-mode>
  <properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:dataPU_embedded;create=
    <property name="javax.persistence.jdbc.password" value=""/>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="javax.persistence.jdbc.user" value=""/>
    <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
  </properties>
</persistence-unit></persistence>
```

Motivation & Problem

- Hur vet man att façaden är klar och kan användas av logiklagret ?
- Om man ändrar i sin façade i efterhand, hur vet man att något inte gått sönder ?
- Hur vet man att det inte finns några buggar i sin façade ?

9/9

0800 Antam started
 1000 " stopped - antam ✓ { 1.2700 9.037 847 025
 1300 (032) MP-MC 1.58267000 9.037 846 995 correct
 (033) PRO 2 2.130476415 4.615925059(-2)
 correct 2.130476415
 Relays 6-2 in 033 failed special speed test
 in relay 10,000 test.

1100 Started Cosine Tape (Sine check)
 1525 Started Multi-Adder Test.

1545 Relay #70 Panel F
 (moth) in relay.

1630 Antam started.
 1700 closed down.

First actual case of bug being found.

Relay 3145
 Relay 3376

JPA - JUnit

- Viktigt att enhetstesta sin façade med Junit
- "Test your interface, not your implementation"
- Under utvecklingen skriver man först test för de funktioner man tror sin façade kommer behöva, sedan skriver man själva funktionerna
- När alla testen kör så vet man att façaden är klar och fungerar
- Test Driven Development (TDD) är en del av eXtreme Programming
- De viktigaste fördelarna är:
 - Systemet är testat och fungerar som tänkt.
 - Endast de funktioner man behöver har utvecklats, dvs lösningen är inte **over-engineered**.
 - Det går snabbt att kontrollera att ändringar i systemet inte har ändrat funktionaliteten (**recession testing**)

JPA - TDD

Realizing quality improvement through test driven development: results and experiences of four industrial teams - Nachiappan Nagappan & E. Michael Maximilien & Thirumalesh Bhat & Laurie Williams - (2008)

(From abstract of paper)

Case studies were conducted with three development teams at Microsoft and one at IBM that have adopted TDD. The results of the case studies indicate that the pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. Subjectively, the teams experienced a 15–35% increase in initial development time after adopting TDD.

JPA – JUnit exempel

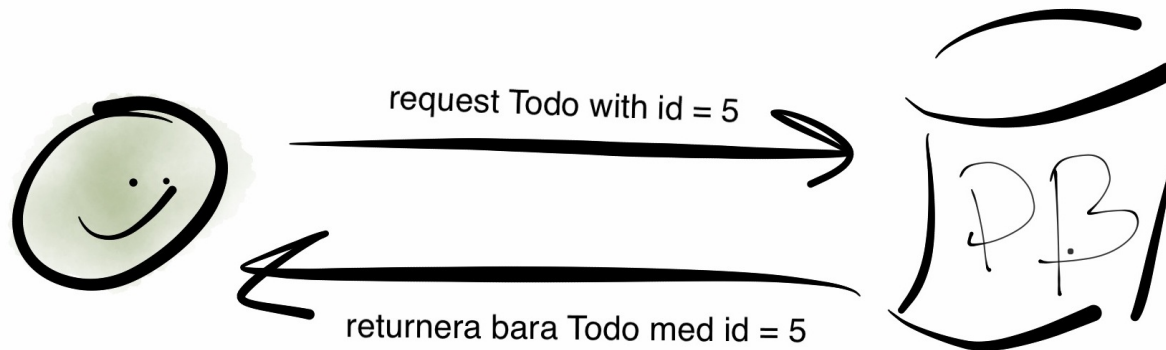
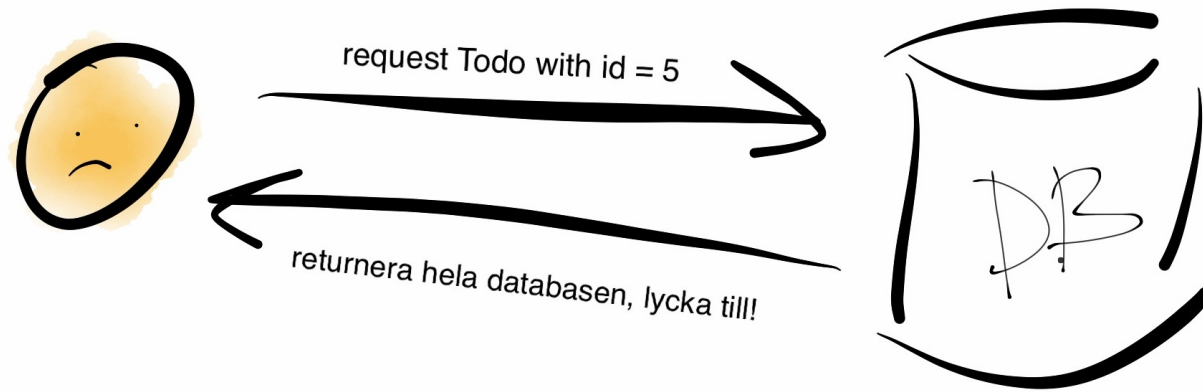
```
@Test
public void test() {

    TodoFacade todoFacade = new TodoFacade();
    long id = todoFacade.create("Todo 1", "Don't forget ...");
    Assert.assertFalse(id == 0);

}
```

Inte ett komplett exempel, men sådana finns i kodskelet och i föreläsnings kod. Det fina med att skriva testfall på detta sätt är att man kan få ut rapporter på hur många av sina test som gått igenom och att man kan få **code-coverage**.

Motivation & Problem



JPA – Find & JPQL

- Vi vill kunna hämta enskilda objekt ur databasen
- Vi vill hitta dessa via valfri column, t ex *id* eller *title*
- Vi vill också kunna hämta alla objekt tillhörande en klass, eller en filtrerad lista
- **JQPL** har ett SQL liknande syntax
- Skapa ett **Query** objekt genom **EntityManager**
- Sätt **parametrar** i Query objekt
- Hämta **ett objekt** eller en **lista**

JPA – Find & JPQL - Exempel

Find by id (@Id)

```
public Todo find(long id) {
```

```
    EntityManager em = EMF.createEntityManager();
```

```
    Todo todo = em.find(TodoDB.class, id);
```

```
    return todo;
```

```
}
```

Interface

Letar efter en implementation

Vi skriver inte till databasen, så vi behöver ingen transaction

JPA – Find & JPQL - Exempel

Find by title

```
public Todo findByTitle(String title) {  
  
    EntityManager em = EMF.createEntityManager();  
    Query query = em.createQuery(  
        "SELECT t FROM TodoDB t WHERE t.title = :titleParam"  
    );  
    query.setParameter("titleParam", title);  
    return (Todo) query.getSingleResult();  
}
```

*Vi skriver inte till databasen, så vi behöver ingen transaction.
Exemplet är inte komplett då vi inte hanterar eventuella fel.*

JPA – Find & JPQL - Exempel

Find all

```
public List<Todo> findAll() {  
  
    EntityManager em = EMF.createEntityManager();  
    Query query = em.createQuery(  
        "SELECT t FROM TodoDB t"  
    );  
    return query.getResultList();  
}
```

*Vi skriver inte till databasen, så vi behöver ingen transaction.
Exemplet är inte komplett då vi inte hanterar eventuella fel.*

Motivation & Problem

“Anything that can go wrong will go wrong”

- Murphy's Law

- Data -> Databas (Nätverk, inkorrekt data, etc)
- Databas -> Data (Nätverk)
- Jordbävning i serverhall ?
- Strömavbrott ?
- “Moth in relay” ?



JPA - Exceptions

- Ibland avbryts exekveringen pga ett fel
- Servrar kan vara nere, nätverk kan vara otillgängliga
- Ett logiskt fel, t ex samma email för flera konton
- Viktigt att hantera olika typer av fel på olika sätt
- Att bara hantera felet "tyst", utan att tala om att något har gått fel är oftast inte så användbart
- Vi vill tala om för anropet att något gått fel, men inte krascha systemet

JPA – Exceptions - Exempel

```
public class TodoEntityFacadeDB implements TodoEntityFacade {

    public long create(String title, String body)
        throws Exception {

        if(title == null || title.isEmpty()) {
            throw new IllegalArgumentException(
                "Title can't be null or empty"
            );
        }

        ...

    }

}
```

Återblick – Façade, EMF, Transactions

```
public class TodoEntityFacadeDB implements TodoEntityFacade {  
  
    public long create(String title, String body) {  
        EntityManager em = EMF.createEntityManager();  
  
        em.getTransaction().begin();  
  
        Todo todo = new TodoDB();  
        todo.setTitle(title);  
        todo.setBody(body);  
  
        em.persist(todo);  
        em.getTransaction().commit();  
  
        return todo.getId();  
    }  
}
```

JPA – Exceptions - Exempel

... samma kod som tidigare exempel ...

```
EntityManager em = EMF.createEntityManager();
```

```
try {
```

```
    //samma kod som i tidigare men vi stänger inte EntityManager.
```

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

```
    ... gör något med felet ...
```

```
    return 0; ←
```

```
} finally {
```

```
    if(em.getTransaction().isActive()) {
```

```
        em.getTransaction().rollback();
```

```
    }
```

```
    em.close();
```

```
}
```

Returnera 0, då vet användaren att objektet inte har sparats i databasen och att något gått fel.

"finally" körs alltid, oavsett om fel har uppstått eller inte, det ger oss möjlighet att göra rollbacks på transaktionen och stänga vår EntityManager innan vi gör något annat.

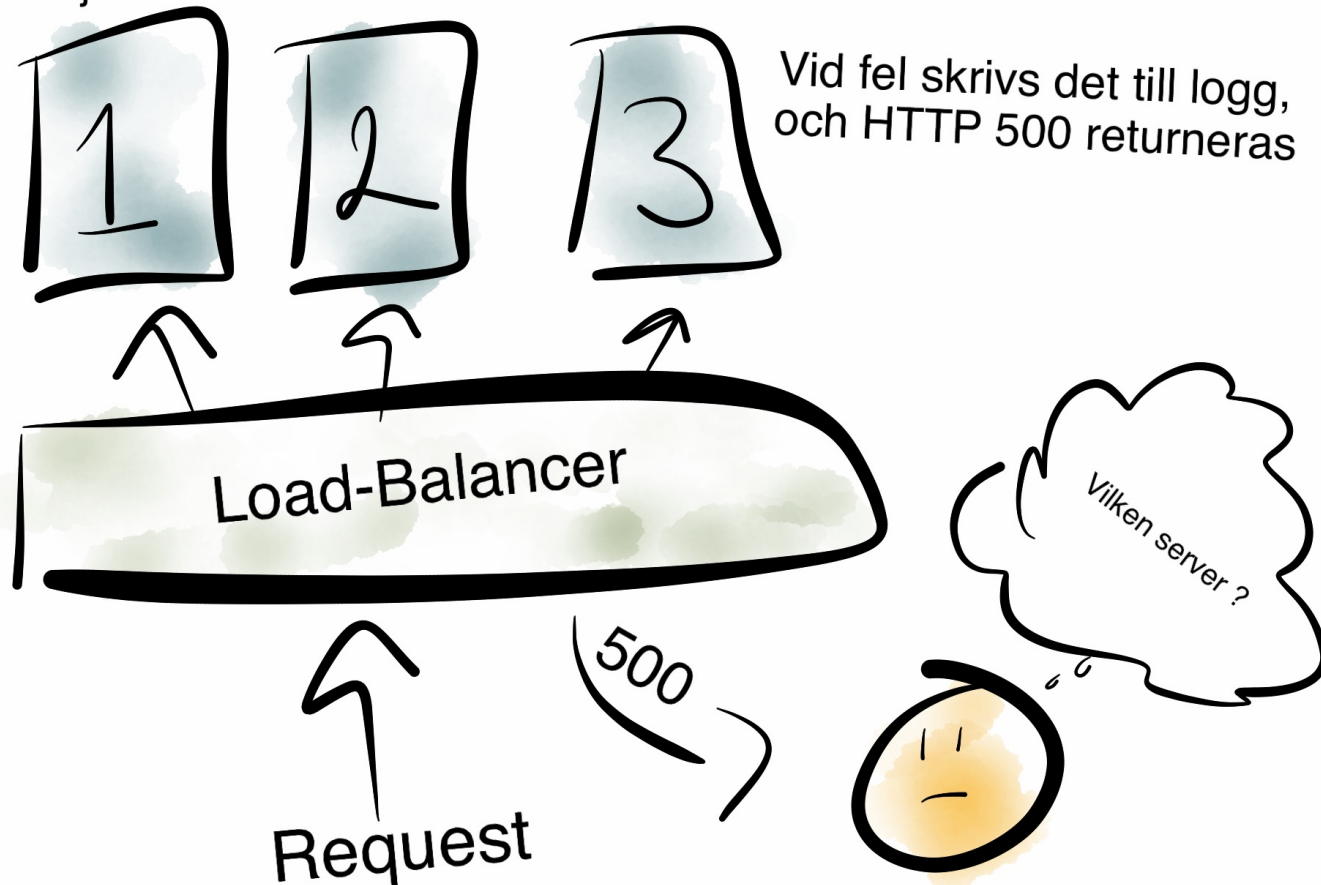
Vi tittar på detta i ett tydligare exempel senare...

JPA - Logging

- Att logga alla anrop och fel gör att det blir enklare att hitta buggar när systemet väl är igång
- Genom att skriva till något persistent, istället för bara System.out, så kan vi gå tillbaka i tiden och även visualisera med HTML eller annat format
- Loggning kan också ske i form av mejl
- Men det finns ett problem med den modell vi visat här
- Vad händer om vår tjänst ligger bakom en load-balancer och är skalad till 20 servrar ?
- Alla anrop hamnar på olika servrar, så då har vi 20 filer som alla har loggat delar av en användares väg genom systemet

Motivation & Problem

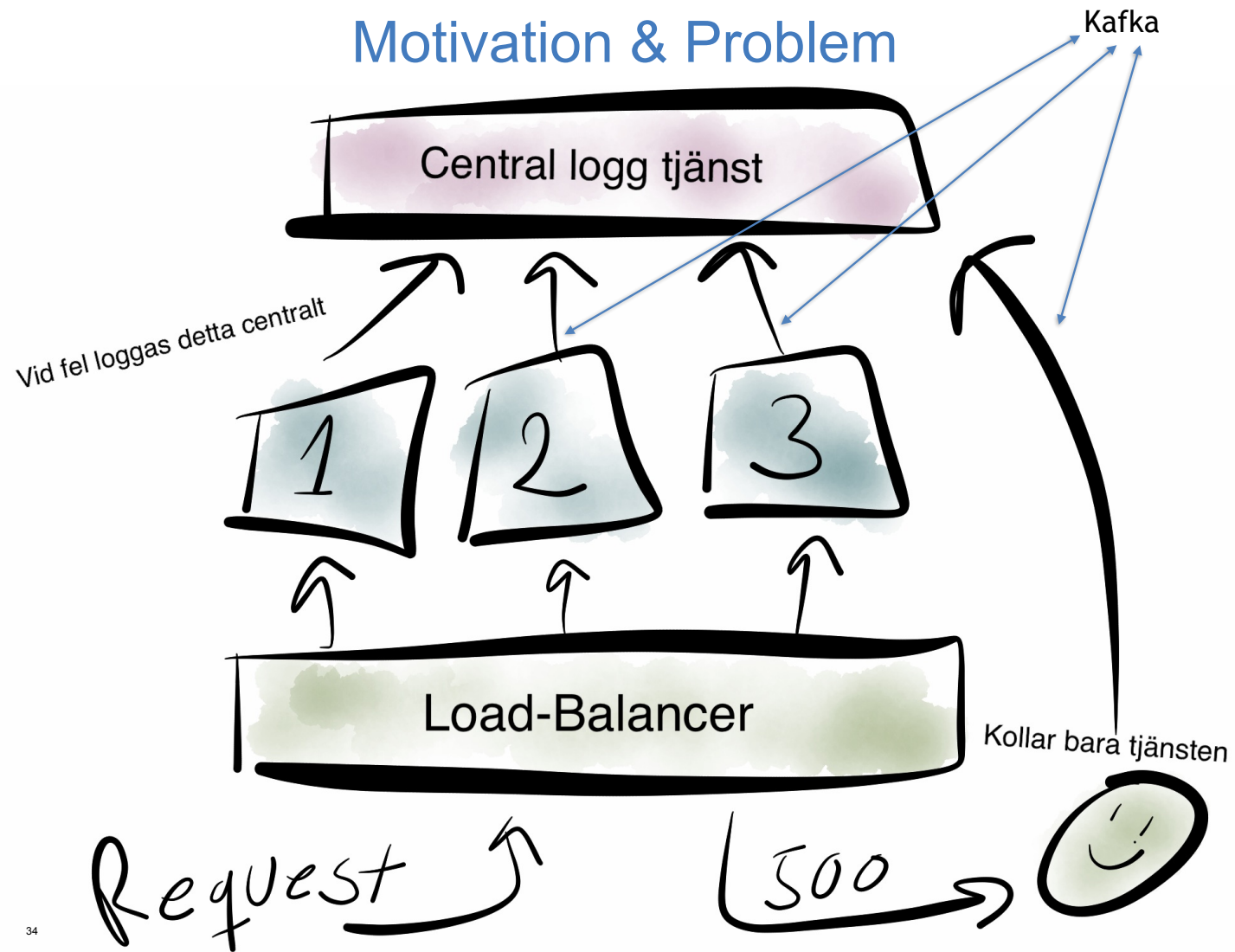
Varje server har var sin lokal loggfil



Distribuerad Loggning

- Istället bör man komplettera sin vanliga loggning med distribuerad loggning
- T ex en tjänst som tar emot loggmeddelanden och ordnar dessa centralt

Motivation & Problem

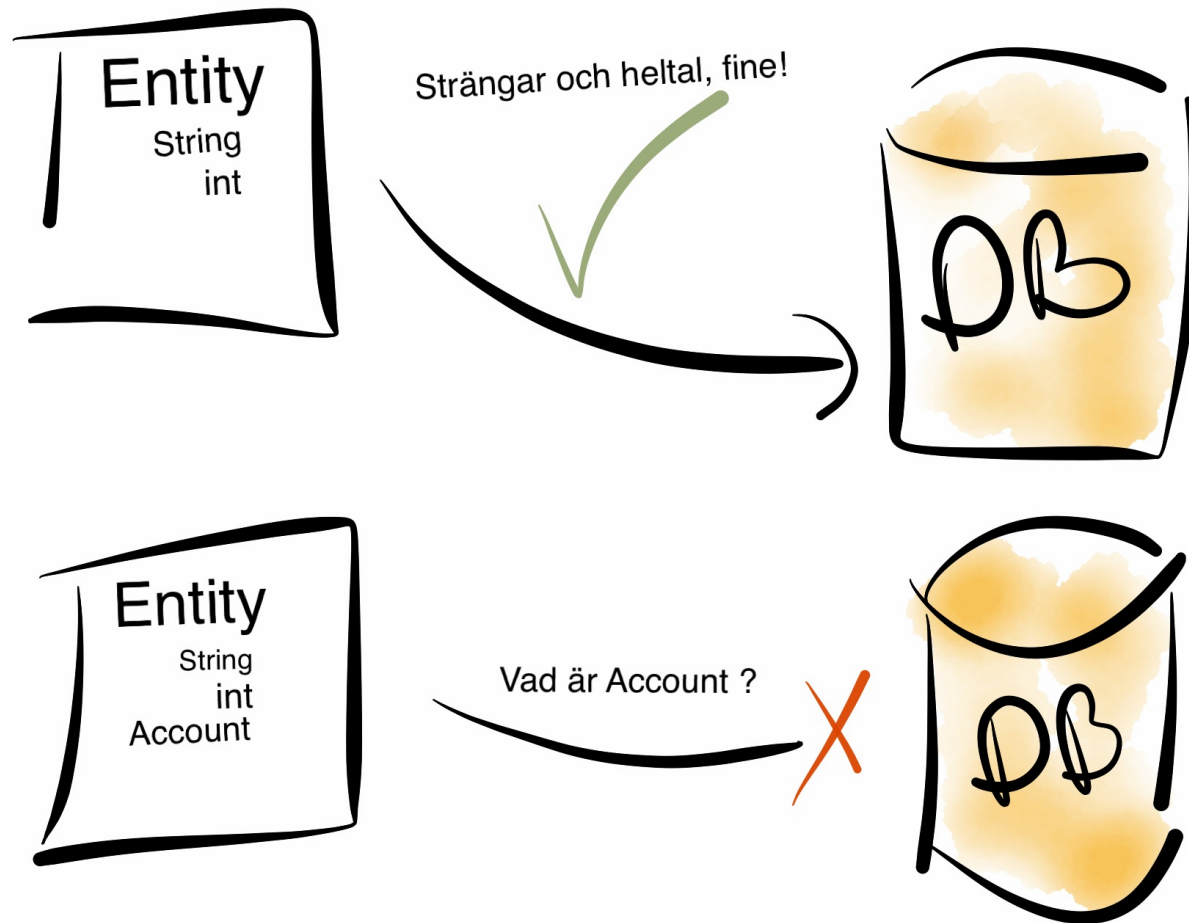


JPA - Mappings

id - bigint(20)	title - varchar(255)	content - varchar(255)
1	Title 1	Content 1
2	Title 2	Content 2
3	Title 3	Content 3
...	...	...
...	...	...

*Vår första Entity skapade en tabell som ovan,
den innehåll bara strängar och heltal ...*

Motivation & Problem



JPA - Mappings

- Komplexa förbindelser, Todo <-> Account
- @OneToMany
@OneToOne
@ManyToOne
@ManyToMany
- Väldigt viktigt med utförliga testfall för att förstå hur dessa "mappings" påverkar varandra och databasens uppbyggnad
- Det finns många inställningsmöjligheter med mappings, och det är ofta viktigt att testa en inställning, titta på hur databasen skapas för detta och ta ett beslut om man vill använda den strukturen
- Skriv testfall som testar vad som händer om man tar bort, lägger till, uppdaterar objekten i en relation

JPA – Mappings exempel

Todo

```
@Entity
public class TodoDB
    implements Todo {

    @Id
    private long id;
    private String title;
    private String content;

    @ManyToOne(targetEntity=CategoryDB.class)
    private Category category;

    ...
}
```

Category

```
@Entity
public class CategoryDB
    implements Category {

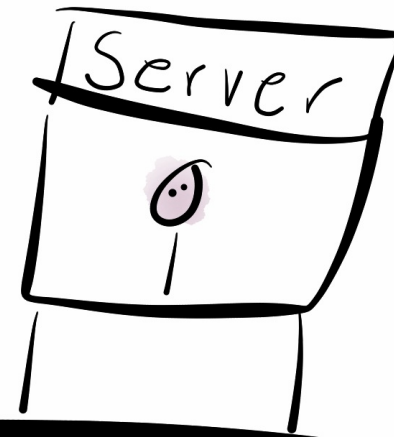
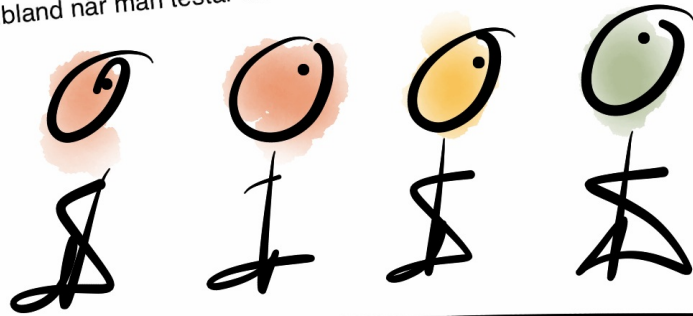
    @Id
    private long id;
    private String username;

    @OneToMany(mappedBy="category",
        targetEntity= TodoDB.class)
    private List<Todo> todos;

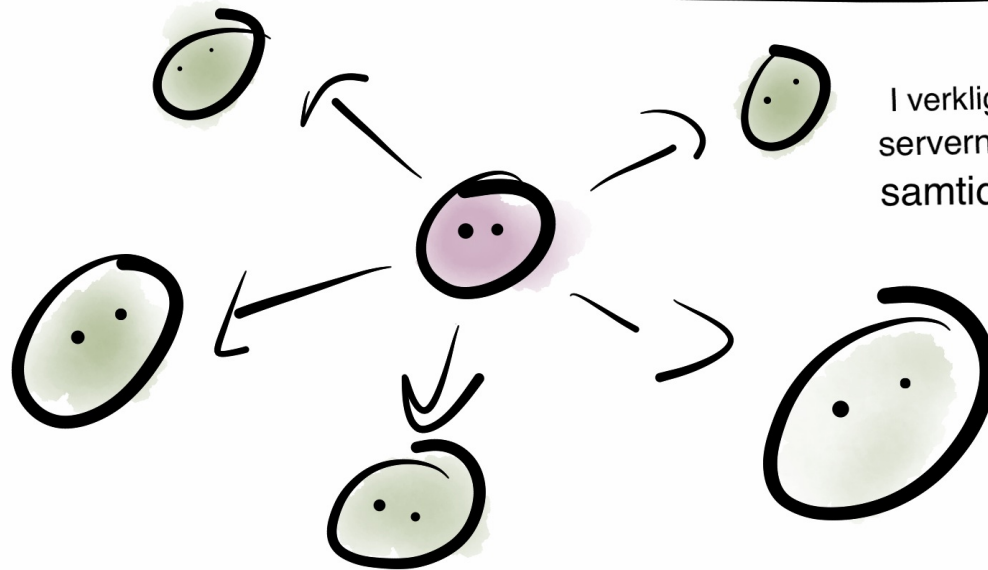
    ...
}
```

①

Ibland när man testar känns det som om servern bara gör en sak i taget



②



I verkligheten hanterar servern flera requests samtidigt.

JPA - Concurrency

- Våra tjänster har inte bara en tråd
- Räkna med att för varje ny användare skapas en ny tråd, och då ofta en ny EntityManager
- Vad händer om man försöker köra samma funktion samtidigt, t ex uppdatera en Todo ?
- **Pessimistisk:** Lås hela raden som skall ändras, ingen annan får röra den tills tråden är klar, lås upp när vi är klara
- **Optimistisk:** Hämta objekt och ändra som vanligt, men om objektet har ändrats i databasen när vi försöker skriva så fallerar funktionen

JPA – Concurrency Exempel

```
...  
public long update(long id, String title, String body) {  
  
    ...  
    Todo todo = em.find(TodoDB.class, id, LockModeType.PESSIMISTIC_WRITE);  
    ...  
}
```

För andra varianter av LockModeType kolla länkar under kurslitteratur på hemsidan

JPA – Concurrency – Klassiskt exempel

(OBS: Pseudo kod)

```
function debit(amount) {  
  
    Account account = findAccount();  
  
    if(account.credit >= amount) {  
        account.credit = account.credit - amount;  
        persistToDatabase(account);  
        return true;  
    } else {  
        return false;  
    }  
  
}
```

Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr

JPA – Concurrency – Klassiskt exempel

(OBS: Pseudo kod)

Tråd 2 väntar....

```
function debit(amount) {  
  
    Account account = findAccount();  
  
    if(account.credit >= amount) {  
        account.credit = account.credit - amount; ← Tråd 1  
        persistToDatabase(account);  
        return true;  
    } else {  
        return false;  
    }  
  
}
```

Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr

JPA – Concurrency – Klassiskt exempel

(OBS: Pseudo kod)

```
function debit(amount) {
```

```
    Account account = findAccount();
```

```
    if(account.credit >= amount) {
```

```
        account.credit = account.credit - amount;
```

```
        persistToDatabase(account);
```

```
        return true;
```

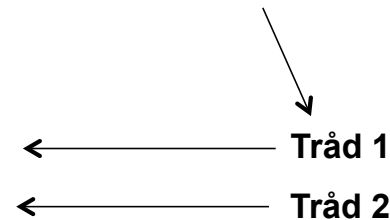
```
    } else {
```

```
        return false;
```

```
    }
```

```
}
```

CPU'n har valt att arbeta med tråd 2, tråd 1 står därmed still.



Tråd 2 debiterar och skriver till, databasen. Account har nu 0kr på sitt konto.

Antag att Account bara har 100kr på sitt konto, och vi får in två requests samtidigt med en debitering på 100kr

JPA – Concurrency – Klassiskt exempel

(OBS: Pseudo kod)

```
function debit(amount) {  
    Account account = findAccount();  
  
    if(account.credit >= amount) {  
        account.credit = account.credit - amount;  
        persistToDatabase(account);  
        return true;  
    } else {  
        return false;  
    }  
}
```

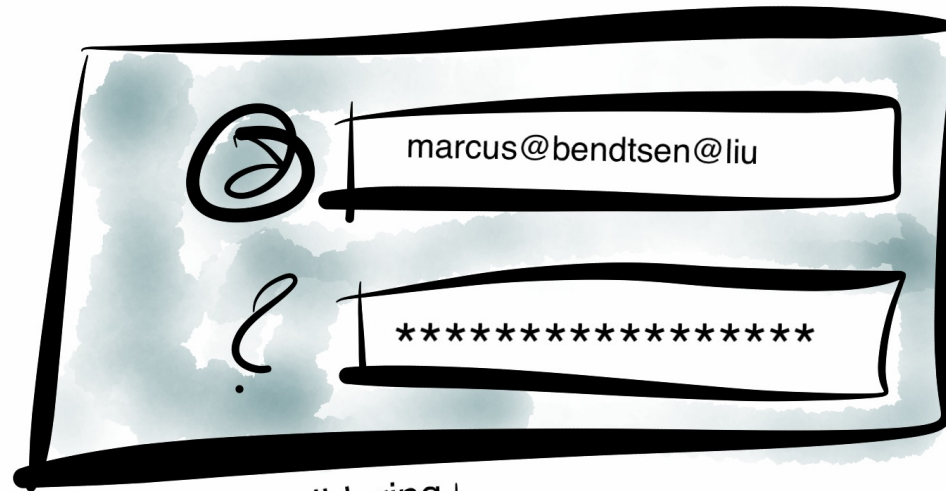
Tråd 2 är klar...

Tråd 1 har redan kollat att
account hade ≥ 100 kr, så
den kör på.

← **Tråd 1**

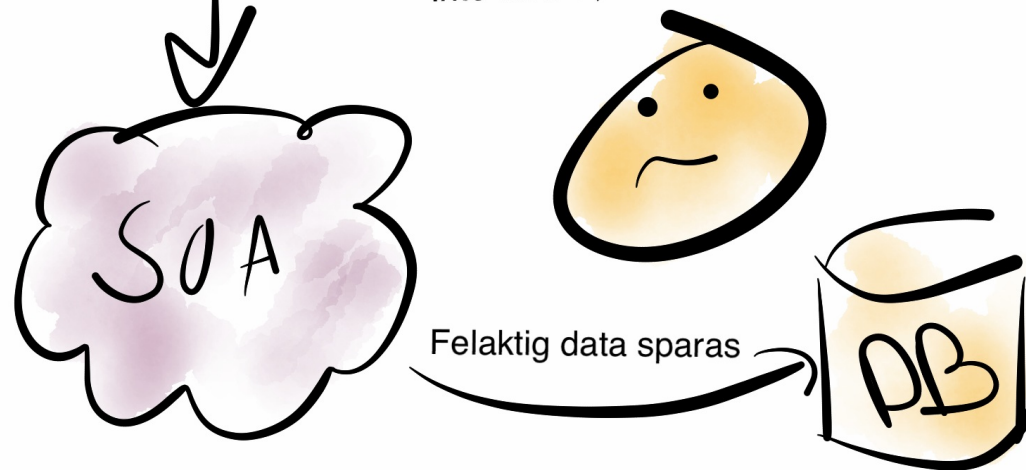
När Tråd 1 är klar har vi debiterat,
200 kr från account (som bara hade
100kr) och `account.credit == 0`

Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr



Ingen JavaScript validering

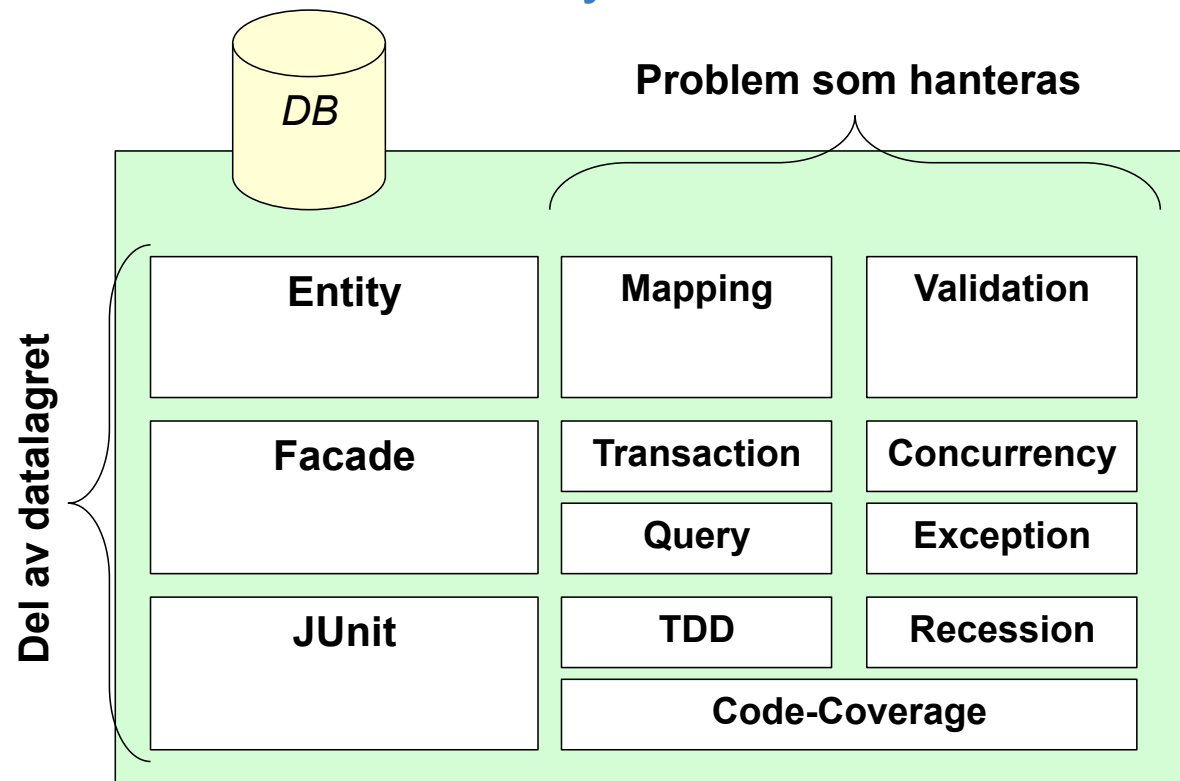
Inte så bra, vart i stacken löser vi detta ?



JPA - Validering

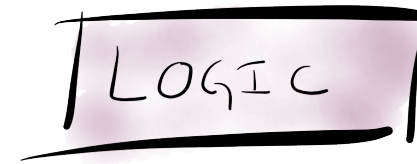
- Det finns bibliotek som del av Java EE som löser detta problem.
- Man kan automatiskt validera email, siffror, max o min, godtyckliga strängar etc.
- Biblioteket kallas Bean Validation.
- Vi kommer inte att ta på oss problemet med validering i denna kurs, men det är naturligtvis ett viktigt problem i större system.

Datalayer - DONE



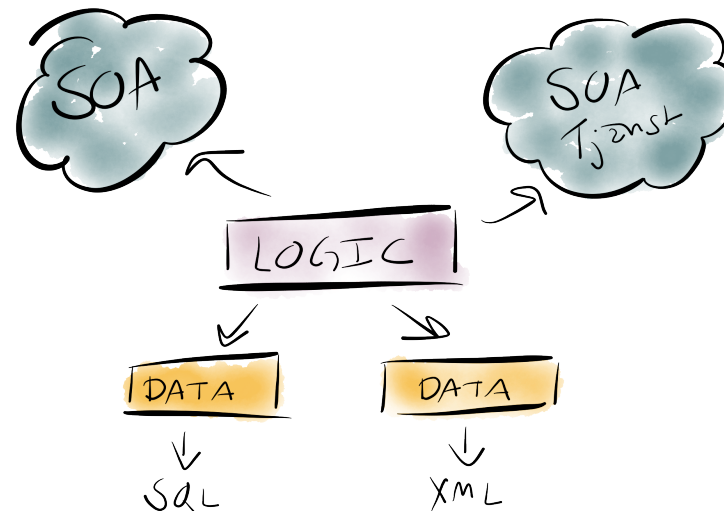
Business Logic Layer

- Business Logic – Kan förklaras som de funktioner som får ”företaget” eller ”systemet” att fungera.
- T ex för en p-automat kan det vara – **printTicket**, **ejectChange** etc. För en webbplats kan det vara **registerUser**, **loginUser** etc.
- Det är de funktioner som är av intresse för omvärlden att använda sig av, som ger värde till företaget, därav "business logic"



Business Logic Layer

- För att skapa värde till företaget behöver logiklagret göra två saker:
- 1. Anrop till externa tjänster inom och utom företaget. Man kan behöva anropa t ex openID för inloggning, skapa Git konton, skicka faktura, skriva till Twitter etc....
- 2. Kommuniera med datalagret (man skulle kunna ha fler datalager i samma SOA tjänst)



Data vs Logic

- I datalagret har man: create(...)
- I logiklagret har man: register(...)
- **register** kommer behöva **create**, men funktionen kanske också måste göra en hel del annat.
- Att skriva till databasen är ett steg i processen, men det kanske behövs ett Git konto, Apache server, etc. som måste skapas, det kan inte datalagret.
- Datalagret kan returnera data till logiklagret, men kan inte anropa logiklagret. Datalagret har inte ens tillgång till logiklagret vid kompilering, så man kommer inte så långt med ett sådant arbete.

DATA

V.S.

LOGIC

Logic Layer – Externa tjänster

- Hur man anropar externa tjänster är oftast olika beroende på vad man anropar (Twitter, Facebook, Google, Amazon etc).
- I kodskelettet finns det en fil HTTPHelper som kan användas för att enklare göra GET anrop till servrar.
- Oftast får man titta i dokumentationen för tjänster hur man gör, men ibland så finns det kompilerade bibliotek som man bara inkluderar i sitt system och sedan kan man anropa tjänsterna som om de var vanliga Java funktioner (t ex Monlog).



HTTPHelper

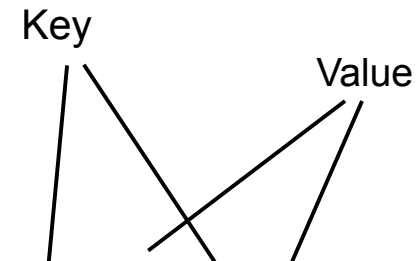
Antag att man vill anropa:

<http://www.google.com?name=M&q=B>

Då anropar vi med HTTPHelper:

```
HTTPHelper helper = new HTTPHelperImpl();
```

```
String response = helper.get("http://www.google.com", "name", "M", "q", "B");
```



Strängen response innehåller nu den text som tjänsten returnerade, i ert fall oftast JSON. Tänk på att tjänsten kan returnera strängen "null" vid vissa fel, inte Java värdet null.

JSONSerializer

```
TodoJsonSerializer jsonSerializer = new TodoJsonSerializerImpl();
```

```
Todo todo = todoLogicFacade.find(id);
```

```
String json = jsonSerializer.toJson(todo);
```



Detta är nu en JSON representation av Todo

```
public class TodoDTO {  
    private String content;  
    public TodoDTO() {}  
    public String getContent() { return content; }  
    public void setContent(String content) { this.content = content; }  
}
```

DTO – Data Transfer Object, ett vanligt *design pattern* som man använder hela tiden utan att tänka på det...

```
TodoJsonSerializer jsonSerializer = new TodoJsonSerializerImpl();
```

```
String json = "{\"content\" : 'hello'}";
```

← **Detta skulle kunna komma från ett anrop via HTTPHelper.**

```
TodoDTO todo = jsonSerializer.fromJson(json, TodoDTO.class);
```

```
System.out.println(todo.getContent());
```

Logic Layer - Exempel

```
public class TodoLogicFacadeImpl implements TodoLogicFacade {
```

```
    private TodoEntityFacade todoEntityFacade;
```

```
    ....
```

```
    public Todo checkOut(long id) {
```

```
        try {
```

```
            Todo todo = todoEntityFacade.find(id);
```

```
            todoEntityFacade.updateStatus(todo.getId(), true);
```

```
            Twitter4j.tweet(todo.getTitle() + " was checked out");
```

```
            return todo;
```

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

```
            ... ta hand om felet ...
```

```
            return null;
```

```
        }
```

```
    }
```

```
}
```

Returnera null om något misslyckas



Logic Layer - Exceptions och JUnit

- I logiklagret kan det också ske fel, kanske ännu fler än i datalagret.
- Tjänster man anropar kanske inte svara, anrop till datalagret lyckas inte, etc.
- Likaväl som man testar sitt datalager med JUnit så skall man testa sin logik med JUnit
- Testningen går till på exakt samma sätt som i datalagret
- Det är lika viktigt här med code-coverage och att man använt TDD för att inte *over-engineer* systemet

Logic Layer – Consistency och Rollback

- Något som är väldigt svårt att generalisera, eller även definiera är *rollbacks* i logik lagret.
- Antag att vi gör tre steg:
 - Lägg en beställning på vara i databasen.
 - Skicka faktura till kund.
 - Skriv till Twitter att vi sålt en vara.
- Om vi inte lyckas skicka fakturan, hur hanterar vi det?
- Om vi inte lyckas skriva till Twitter, hur hanterar vi det?
- Utanför ramen för denna kurs, men en viktig aspekt.



Linköpings universitet
expanding reality

www.liu.se