# IN2090 – Databaser og datamodellering

## 14 – Sikkerhet

Leif Harald Karlsen
leifhka@ifi.uio.no

Universitetet i Oslo



## Hovedmål med databasesikkerhet

- Konfidensialitet
  - Uvedkommende må ikke kunne se data de ikke skal ha tilgang til
- Integritet
  - Data må være korrekte og pålitelige. Derfor må data beskyttes mot endringer fra uautoriserte brukere.
- Tilgjengelighet
  - Brukere må kunne se eller modifiesere data de har behov for



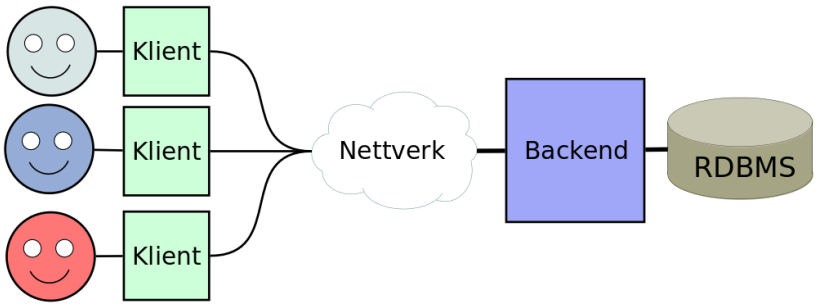
## Sikkerhet: Ikke bare i databasesystemet

- Programmer inneholder ofte mye mer enn bare databasen
- Databasesikkerhet kan derfor ikke kun fokusere på databasen
- Sikkerhetshull kan forekomme i alle ledd (frontend, backend, nettverket, osv.)
- Sikkerhet er derfor alltid en *helhetlig* oppgave
- Må derfor sikre hver enkelt komponent og interaksjonen mellom dem
- Må være tydelige på hvilke antagelser om andre komponenter hver del av systemet gjør



## Sikkerhet: Ikke bare i databasesystemet

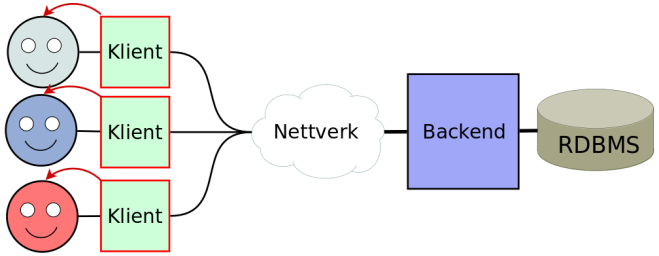
## Oversikt over systemer med databaser






## Tilgangskontroll: Klient/Frontend

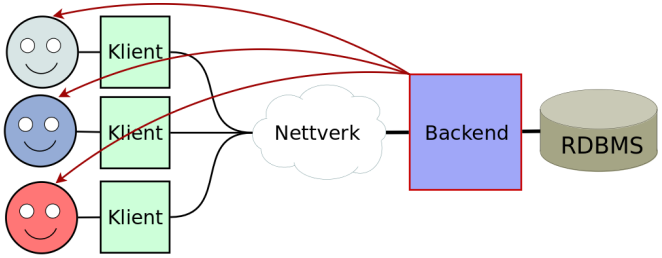
- Klienten autentiserer brukerne
- Klienten sjekker hva brukeren har lov til
- Backend og RDBMS må stole på at klienten gjør dette riktig
- Trenger sikring slik at bare klienten kan få tilgang til backend/RDBMS






## Tilgangskontroll: Backend

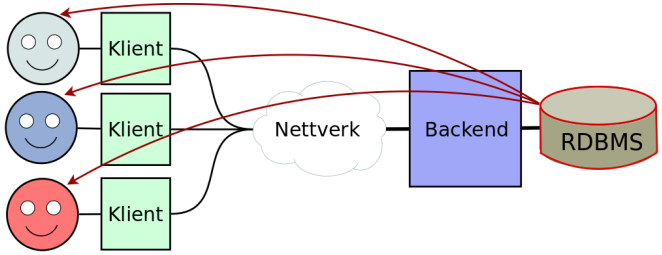
- Backend autentiserer brukerne
- Backend sjekker hva brukeren har lov til
- Backend har ofte én bruker til databasen
- RDBMS må stole på at backend gjør dette riktig
- Backend og RDBMS ofte bak samme brannmur






## Tilgangskontroll: Database

- Databasesystemet autentiserer brukerne direkte
- Hver bruker av programmet får da hver sin databasebruker
- Klienten kan forhåndssjekke (f.eks. for å tilpasse brukergrensesnittet)



## Tilgangskontroll i databaser

- Tilgang til databasen kontrolleres gjennom tre ting:
  - brukere
  - roller
  - rettigheter
- For eksempel:
  - Bruker `leifhka` har rollen `kundeadmin`
  - Bruker `leifhka` har rollen `produktansvarlig`
  - Bruker `klient` har rollen `kunde`
  - Brukere med rollen `kundeadmin` kan opprette og oppdatere kunder (rader i `customer`-tabellen)
  - Brukere med rollen `produktansvarlig` kan opprette, oppdatere og slette produkter (rader i `products`-tabellen)
  - Brukere med rollen `kunde` kan se på produkter (rader i `products`-tabellen) samt legge inn ordre (rader i `orders`-tabellen)



## Brukere vs. roller

- Mulig å gi hver bruker de rettighetene de skal ha
- Men vanskelig å holde rede på at hver bruker har de riktige rettighetene
- Spesielt om det er mange brukere og mange rettigheter
- Typisk vil mange brukere trenge samme rettigheter: Vanskelig å vedlikeholde
- Vi lager derfor roller som fanger en mengde med rettigheter som hører sammen
- Og gir deretter brukere de passende rollene
- Dette heter *Role-based Access Control*



## Databasebrukere

- Uansett hvilken autentiseringsmodell man bruker må man ha databasebrukere ("user")
- F.eks. når dere logger dere inn i databasen med:

  ```
  $ psql -h dbpg-kurs-ifi -U leifhka -d fdb
  ```

  er `leifhka` brukeren
- Autentisering skjer typisk via passord, SSH public keys, el.
- Autentisering kan delegeres til andre systemer
- Gydige brukerkontoer og (krypterte) passord lagres av RDBMS
- Alle databaser, skjema, tabeller, views, osv. eies av en bruker
- Men eierskap betyr ikke nødvendigvis rettigheter
- Alle transaksjoner kan spores til brukeren som eksekverte dem



## Lage brukere og roller med SQL

- For å lage en ny bruker `leifhka` med passord `hemmelig` og rollene `kundeadmin` og `produktansvarlig` kan man kjøre følgende SQL-kommando[1]

  ```
  CREATE USER leifhka WITH PASSWORD 'hemmelig'
      ROLE kundeadmin, produktansvarlig;
  ```

- Roller lages nesten helt likt[2]:

  ```
  CREATE ROLE produktansvarlig;
  ```

- I PostgreSQL er `CREATE USER` bare et alias for `CREATE ROLE` med `LOGIN`-adgang (mao. brukere er bare en spesiell type rolle)
- Roller og brukre slettes med `DROP`
- Merk: Som oftest bare superbrukere som kan lage brukere/roller

[1] se `https://www.postgresql.org/docs/12/sql-createuser.html`
[2] se `https://www.postgresql.org/docs/12/sql-createrole.html`

## Begrense bruk

- Kan bergrense hvor lenge en bruker eller rolle skal være gyldig ved å sette `VALID UNTIL '2020-01-01'` i kommandoene over
- Kan begrense antall tilkoblinger en bruker/rolle kan ha ved å sette `CONNECTION LIMIT 5` i kommandoene over
- Dette er det som gjør at noen av dere har fått feilmeldingen:

```
psql: FATAL:  too many connections for role "user_name"
```

- For å gi en bruker/rolle (generelle) rettigheter til å lage databaser, roller, osv. kan man legge til `CREATEDB`, `CREATEROLE`, osv.



## Gi og fjerne rettigheter

- Man kan gi roller/brukere mer detaljerte rettigheter via `GRANT`-kommandoen[3]
- `GRANT`-kommandoen har følgende form:

```
GRANT <privileges> ON <object> TO <role>;
```

- hvor `<privileges>` f.eks.:

```
SELECT, UPDATE, INSERT, DELETE, CREATE, CONNECT, USAGE, ALL
```

- og `<object>` er f.eks. en database, en tabell, et skjema, el.
- Gir man rettigheter til en rolle, vil alle dens medlemmer også få disse
- Fjerning av rettigheter kan gjøres tilsvarende med `REVOKE`

[3] `https://www.postgresql.org/docs/12/sql-grant.html`



## GRANT-eksempler

- For å gi rollen `kundeadmin` rettighetene til å oprette og oppdatere `ws.users`-tabellen kan vi kjøre følgende kommando:

```
GRANT INSERT, UPDATE ON ws.users TO kundeadmin;
```

- For å gi rollen `webshopadmin` alle rettigheter innenfor skjemaet `ws`:

```
GRANT ALL ON SCHEMA ws TO webshopadmin;
```

- Kan også gi en bruker en ny rolle med `GRANT`:

```
GRANT kundeadmin TO leifhka;
```

- Kan til og med gi tillatelser på kolonnenivå:

```
GRANT UPDATE (price) ON ws.products TO prisansvarlig;
```

- For å fjerne `kunde`-rollens tilgang til `categories` kan vi kjøre

```
REVOKE USAGE ON ws.categories FROM kunde;
```



## Tilgang og views

- I enkelte tilfeller ønsker vi ikke gi tilgang til tabellene direkte
- Men f.eks. kun aggregerte eller utvalgte verdier
- F.eks. vil ikke gi ut lønnsinformasjon om hver enkelt person, men heller gjennomsnittsinntekt per bydel
- Kan da lage views, og så gi tilgang til disse
- Går også an å lage views man kan oppdatere, men det er utenfor pensum
- Også mulig å spesifisere tilgang på radnivå i tabeller, men er utenfor pensum

## Databasetilkoblinger

- `Connection`-objektene (og de tilhørende `cursor` eller `Statement` og `ResultSet`) er ressurser som kan føre til minnelekasjer
- Alle disse har en egen metode som heter `close()` som stenger ressursen og frigjør den
- For de enkle programmene vi så sist uke var dette ikke veldig viktig, ettersom ressursene blir frigjort når programmet avsluttes
- Men for større programmer og tjenester som skal kjøre over lengre tid bør man alltid sørge for at tilkoblingene blir stengt med en gang man er ferdige med dem



## Stenge tilkoblinger

- I Python har både `Connection` og `Cursor` en `close()` metode
- I Java har både `Connection`, `Statement`/`PreparedStatement` og `ResultSet` egne `close()` metoder
- Generelt blir alle objekter stengt når objektet det er laget fra stenges
- F.eks. vil en `PreparedStatement` stenges dersom dens `Connection` stenges



## Automatisk stenge ressurser

- Java kan bruke `try-with`-blokker (fom. Java 7) for automatisk å stenge tilkoblinger
- F.eks.:

```
try (Connection con = DriverManager.getConnection(conStr);
    PreparedStatement stmt = con.prepareStatement(query)) {
    try (ResultSet res = stmt.executeQuery()) {
        // Do stuff with the result set.
    }
} catch (...) {
    // errors
} // con, stmt, res closed here
```

- Har tilsvarende i Python:

```
with psycopg2.connect(dsn) as conn:
    with conn.cursor() as cur:
        cur.execute(sql)
```

men dette stenger ikke tilkoblingen (kun eventuelle åpne transaksjoner)



## SQL injections

- SQL injection er en type angrep mot en klient/backend hvor en (ondsinnet) bruker får kjørt egen SQL-kode på databasen
- Brukeren kan da forbigå autentisering eller hente ut, endre eller slette data brukeren egentlig ikke skal ha tilgang til
- SQL injection utnytter følgende faktorer:
  - At vi har et program som kjører SQL-spørringer (f.eks. Java eller Python)
  - Programmet tar input fra brukeren som skal være en del av spørringene
  - Programmet ikke skiller mellom data og kommandoer i spørringen

## SQL injections: Grunnleggende prinsipp

- La oss si at en nettbutikk lar brukere søke etter varer på følgende måte:

```
product = input("Product: ");
cur.execute("SELECT * FROM products WHERE navn = '" + product + "';");
```

- Med input `Socks` blir spørringen:

```
SELECT * FROM products WHERE navn = 'Socks';
```

- Med input `O'Boy` får vi error:

```
SELECT * FROM products WHERE navn = 'O'Boy';
```

- Med input `Socks'; DROP TABLE products;--` får vi

```
SELECT * FROM products WHERE navn = 'Socks'; DROP TABLE products; --';
```

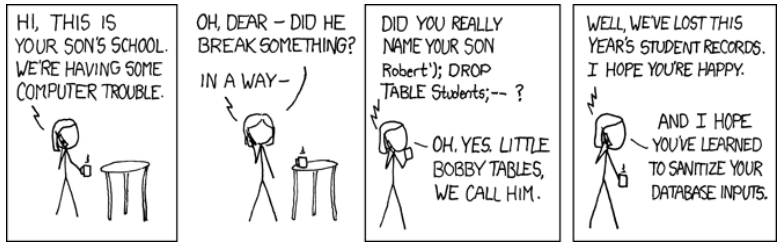


## SQL injections: Webshop-eksempel

Live-kode-eksempel!



## XKCD




`https://xkcd.com/327/`

Hvis du ikke skjønte den:

`https://stackoverflow.com/questions/332365/how-does-the-sql-injection-from-the-bobby-tables-xkcd-comic-work`



## Hvorfor er dette viktig?

- En av de vanligste formene for hackerangrep
- Dersom angrepet lykkes vil det gi den ondsinnede brukeren veldig mye makt:
  - Ødelegge/slette data
  - Endre data
  - Hente ut konfidensiell informasjon
  - Hindre tjenesten i å fungere (DOS)
- Veldig enkelt å forhindre med parametriserte spørringer!

## Helhetlig sikkerthet

- Merk at selvom klienten skulle være sårbar for SQL injection-angrep kan databasen likevel forhindre mye skade om man har satt riktige rettigheter
- F.eks. dersom databasebrukeren som klienten benytter ikke har tilgang til å slette eller endre tabeller eller spørre mot vilkårlige tabeller
- Dette viser hvor viktig det er at alle ledd er sikret og at man ikke bør anta for mye av andre komponenter i et system