# Inloggning (autentisering)

TDDD80 Mobila och sociala applikationer

LINKÖPINGS UNIVERSITET

# Två deluppgifter

1. Spara lösenord på ett säkert sätt
   - I databas på server-sidan
2. Hålla koll på inloggningsstatus
   - Delegeras till varje klient
     - Varför?

# Varför: Viktigt att kunna skala upp

- Vill kunna skala upp till många samtidiga användare, … och flera servrar
  - Annan server ska kunna ta hand om ett anrop, även om servern inte har hela historien med just den klienten
    - Klienten håller reda på vad som har hänt hittills
  - Varje anrop från klienten måste vara självtillräcklig, dvs. innehålla all nödvändig information

# REST

- REST (**RE**presentational **S**tate **T**ransfer)
  - Klientens anrop innehåller information om var man är i transaktionen, dvs. vilket tillstånd (state) man är i
    - T.ex. om klienten är inloggad
    - Klienten skickar med ”bevis” på att användaren är inloggad

# På labben

# Ett par nya views för reg och inloggning

```
@app.route('/register', methods=['POST'])
def register_user():
    … # lägg in nytt email, lösenord i DB


@app.route('/login', methods=['POST'])
def login_user():
    … # kolla om skickat lösenord matchar lagrat lösenord
```

# User

- User-tabell utökas med några kolumner

```
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(5))
    email = db.Column(db.String(20) , unique=True)
    passw_hash = db.Column(db.LargeBinary())
```

# Viktigt att lösenordet skyddas

- Vill transformera lösenordet
  - Ska inte kunna avkodas av utomstående
    - bcrypt, scrypt, PBKDF2, Argon2
- Lättläst intro: https://code.tutsplus.com/tutorials/understanding-hash-functions-and-keeping-passwords-safe--net-17577

# Hash-funktioner, enkelt exempel

- In: lösenord på *n* bokstäver
- Ut: summan av ingående ASCII-koderna * 100
- Stort spann på output + kombinatoriskt svårt att reversera
  - Hur fick jag summan 987454?
  - Hur många bokstäver ingick?
  - Vilka bokstäver?

# Hashning

- Stort ut-span försvårar reversering
- Dessutom brukar hashen förstärkas genom att köra *hash(hash(hash(hash…(lösenord))))*

# Attacker

- Även om någon vet hashfunktionen
- Kan bara köra brute-force (blind) attack
  - Antag att någon fått tag på databasen
  - Itererar igenom alla tänkbara lösenord, och ser om *hash(lösenord) = lagrat lösenord*

# Attacker i praktiken

- Väldigt många tänkbara lösenord,
  tar väldigt lång tid att prova alla …

# Regnbågstabeller

- För att hinna på rimlig tid (timmar el. dagar)
  - Förberäknar med kända hash-funktioner på vanliga lösenord
    - T.ex. bcrypt(”password”), bcrypt(“passw”), bcrypt(“1234”), …
    - T.ex. löpa igenom ett engelskt lexikon

- Hur kan man stoppa denna typ av attack?
  - Användare väljer ju oftast enkla lösenord…
  - Vad kan man göra som server-admin för att göra lösenordet svårare att förutsäga?
  - Google-sökning på labben…

# Hasha lösenord på labben

# Ett par nya views för reg och inloggning

```
@app.route('/register', methods=['POST'])
def register_user():
    … # lägg in nytt email, lösenord i DB


@app.route('/login', methods=['POST'])
def login_user():
    … # kolla om skickat lösenord matchar lagrat lösenord
```

# Lagt till extra kolumner i databasen

```
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(5))
    email = db.Column(db.String(20) , unique=True)
    passw_hash = db.Column(db.LargeBinary())
```

# Registrera användare

```
class User(db.Model):
    …
    def __init__(self, name, email, password):
        self.name = name
        self.email = email
        self.pw_hash = some_lib.hash(password)
```

```
def register_user(username, email, password):
    user = User(username, email, password)
    db.session.add(user)
    db.session.commit()
```

skapa ny anv.

# Flera bra bibliotek för lösenordshashning

- Flask-bcrypt (https://flask-bcrypt.readthedocs.io/en/latest/)
- Itsdangerous (http://pythonhosted.org/itsdangerous/)
- Verkzeug (https://werkzeug.palletsprojects.com/en/latest/)

**from** flask_bcrypt **import** Bcrypt

eller

**from** werkzeug.security **import** generate_password_hash,
check_password_hash

# Inloggningsstatus

# Autentisering (= inloggning)

- Server
  - Registrering och inloggning
    - Email, lösenord
- Klient
  - Inloggningsstatus (inloggad)

# Autentiseringsflödet

**klient** **server**

logga in ⟶ anv.namn + lösenord

kollar mot lagrad hash

access token ⟵

⟶ post(...,
+ access token)

# REST (Representational State Transfer)

- Viktig aspekt av REST
  - Servern kommer *inte* ihåg sin interaktion med kilenter från tidigare request-anrop
- Klienten
  - Håller reda på token från servern
    - Sparar den och skickar med vid varje nytt anrop

# Hur request skickas för inloggning

```
response = requests.post(URL_root + '/login',
                         json={'email': user_email,
                               'password': user_pw})
```

# Ett http-request skickas

```
POST /resource/1 HTTP/1.1
Host: example.com/login/
Content-Type: application/json; charset=UTF-8

{
 'email': user_email,
 'password': user_pw
}
```

header

body

# Serverns svar på inloggningsrequest

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{ "access_token": ” mF8hf.Bfsld5f4g.AJqfh ",
"token_type": ”Bearer",
"expires_in": 3600,
"refresh_token": "tGzv3.JOkF0XG5Q.x2TlKWIA"
}

# Klient: lagrar och använder access token

```
response = requests.post(URL_root + '/login',
                         json={'email': user_email,
                               'password': user_pw})

access_token = response.content…

r = requests.post(URL_root + '/messages/',
                  headers={'Authorization' : 'Bearer ' + access_token},
                  …)
```

Plocka fram access token ur response

Skickas med i request-headers vid anrop där inloggad-status krävs

# http-request med access token skickas

POST /resource/1 HTTP/1.1
Host: example.com/message
Authorization: Bearer mF8hf.Bfsld5f4g.AJqfh

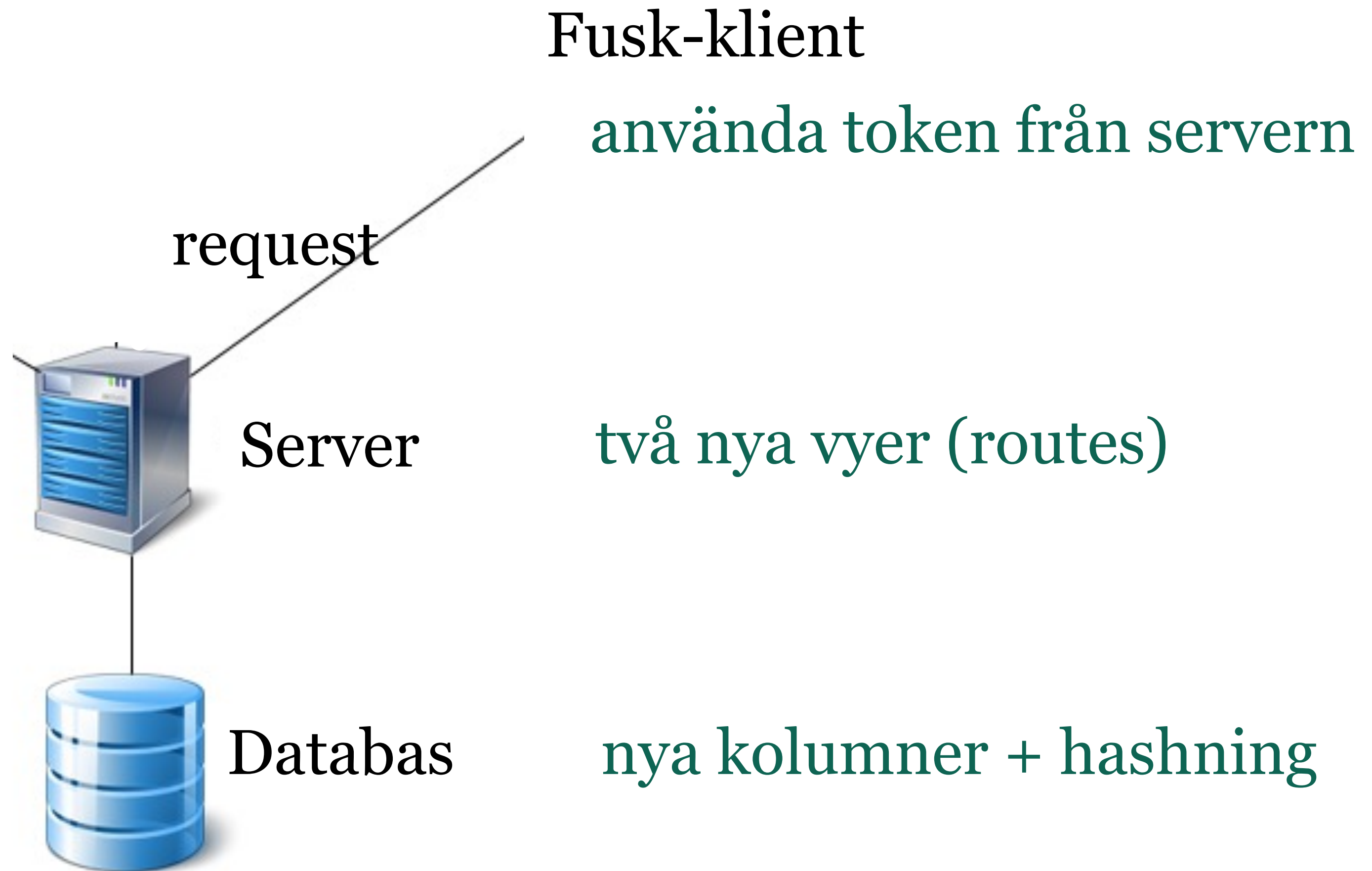
header

{
{'message': 'Hej'}
}

body

# Skapa request med headers

Läs mer på: http://docs.python-requests.org/en/master/user/quickstart/


Fusk-klient
använda token från servern
request
Server
två nya vyer (routes)
Databas
nya kolumner + hashning

# Säkerhetsaspekter av tokens

# Säkerhetsaspekter

- Tänk om klienten missbrukar token
  - Behåller den alltför länge
  - Säljer den till någon annan
  - Fifflar med den, t.ex. ändrar dess innehåll
- Tänk om någon annan kommer över token

# Servern signerar access token

- Skiver under med unik server-‘*secret*’
  - *Secret* ska bara servern ha tillgång till
- Bakar in annan kontroll-information
  - “bäst-före”
  - anv.email,
  - etc.

- Packar ihop allt och hashar, med secret som nyckel

# Token är signerad

- Mottagaren (klienten) vet inte *secret*
  - Kan inte ändra utan att det märks på server-sidan

- För varje request kollar sedan servern info i medskickad access-token
  - Stämmer information i token?
  - Är den fortfarande giltig?

# JWT (JSON Web Token)

- Token header → kodas till xxxxx
- Token payload → kodas till yyyyy
- xxxxx.yyyyy + secret → hashas till zzzzz

T.ex. { "alg": "HS256", "typ": "JWT" }

T.ex. användarinfo

Signature

- JWT:n blir

xxxxx.yyyyy.zzzzz

Innehåller låst kopia på xxxxx och yyyyyy

# JWT claims i JWT-payload (yyyyy)

- Registrerade (standard) jwt ‘claims’

**‘jti’:** jwt_id

**‘iss’:** server_id

**'exp'**: datetime.now() + timedelta(days=0, seconds=50)

**'iat'**: datetime.now()

**'sub'**: user_email

# Ytterligare info i token (private claims)

- Möjlighet att lägga med egna claims
- Vill kanske ha med andra nyckel-värde par
  - T.ex. {‘device’: ‘phone’,
    ’email’: user_email}
  - Fördel med JSON: ej förutbestämda nycklar

# Håll server secret hemligt!

Om man använder biblioteket flask_jwt_extended:

Sätt config-variabler

```
app.config['JWT_ACCESS_TOKEN_EXPIRES'] =
    datetime.timedelta(minutes=500)

app.config['SECRET_KEY’] = os.environ[’SERVER_SECRET']
```

```
>> export SERVER_SECRET = ‘sfjhas fsahf safhasökfjhas fsdf’
```

# Lagra server secret i .env-fil

>> pip3 install python-dotenv

SERVER_SECRET=’kfjga fyt/&€&(%#XC HGF )(Ykjl’

DBHOST='tddd80-lab33-rmk-server'

DBNAME='postgres'

DBPASS=’…'

DBUSER=’…’

# .env-fil

- .env-filen ska **inte** versionshanteras
  - Lägg med ”.env” i er .gitignore-fil:

```
.env
.idea
.venv_3.10
app.db
```

# Läsa server secret från .env

```
if 'WEBSITE_HOSTNAME' in os.environ:   # on Azure
    database = os.environ['DBNAME’]
    ...
else:
    from dotenv import load_dotenv
    load_dotenv()
```

# Secret ska hållas hemligt

- Undvik att lägga secret i server-koden (och repot)
  - Definiera miljö-variabel
    - genom 'export' i terminal
      - Exporterar till senare processer som startas upp i denna terminal
    - .bash_profile (el. motsv.)
    - i PyCharm i Run configurations
    - i .env-fil
    - i Azure > Configuration > Add/Edit application setting
  - Läs in variabeln från miljön varje gång server-koden körs

# Login vyn

```
@app.route('/login', methods=['POST'])
def login_user():
    content = request.get_json()
    device = content['device']
    email = content['email']
    if (database_handler.login_user(email, password)):
        add_claim = {'device': device}
        token = create_access_token(identity=email,
                                    additional_claims=add_claim)
        return token
    …
```

# Skapa access token

Denna funktion får ni auto-magiskt från biblioteket flask_jwt_extended

```
from flask_jwt_extended import  JWTManager,
                                jwt_required,
                                create_access_token,
                                get_jwt_claims,
                                get_jwt
```

# Inloggningsstatus (mha decorators)

# Första ansats att kolla inloggningsstatus

```
@app.route('/messages/<msg_id>/read/<email>', methods=['POST'])
def mark_message_as_read(msg_id, email):
    if <check_access_token>:
        database_handler.mark_as_read_by(msg_id, email)
        return 'Message marked as read by ' + email


@app.route('/messages', methods=['POST'])
def save_message():
    if <check_access_token >:
        ...
        database_handler.store_message(mess_to_store, message_id)
        return str(message_id)
```

# Nackdelar

- Upprepning av kod
  - Ändra på många ställen om *check_access_token()* ändras
- Svårt att få överblick
  - "Vilka views vad det nu som kräver inloggning"?
- Lättare att göra misstag
  - Ligger efterkommande anrop innanför/utanför if-satsen?

# Snyggare med decorators

- En wrapper (lindar in en funktion):
  - Gör förbearbetningssteg, dvs. kollar inloggningsstatus
  - Vidarebefordrar sedan anropet till funktionen

```
@app.route('/messages', methods=['POST'])
@jwt_required()
def save_message():
    …
```

Tar hand om check_access_token, skickar 401, Unauthorized

# flask_jwt-extended biblioteket

- Wrappern jwt_required() är redan definierad

**from** flask_jwt_extended **import** jwt_required

# När du vill hämta egna claims från en token

```
from flask_jwt_extended import jwt_required, get_jwt
```

```
@app.route(……)
@jwt_required()
def my_view(……):
    claims = get_jwt ()
    email = claims['email’]
```

# Utloggning

# Utloggning innan token *exp* har gått ut?

- T.ex. token exp = en vecka
  - Klienten har token i “handen”, kan fortsätta skicka requests även efter utloggning
- Måste hitta ett sätt att återkalla token
- Servern måste hålla reda på vilka tokens som har återkallats

# Utloggningsvy (utloggningsfunktion)

```
@app.route('/logout', methods=['DELETE'])
@jwt_required()
def logout():
    jti = get_jwt()['jti']
    database_handler.revoke_token(jti)
    return 'Access token revoked', 200
```

# Deny list / block list

- Hur ska vår server komma ihåg att en token är återkallad (revoked)?
- Deny list
  - Tokens som är utloggade sparas i en *deny list*
  - Varje inkommande token kollas mot *deny list*, förutom den vanliga kollen av *exp*, etc.

# Deny list

- Extra tabell i er databas (som delas mellan serverinstanser)

```
class DenyList(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    jti = db.Column(db.String(36), nullable=False)

    def __init__(self, jti):
        self.jti = jti
```

# Baka in det i @jwt_required

```
jwt = JWTManager(app)

@jwt.token_in_blocklist_loader
def is_token_in_blocklist(jwt_header, jwt_payload):
    if  jwt_payload['type'] == 'access':
        jti = jwt_payload['jti']
        return database_handler.in_deny_list(jti)
```

Fördefinierad "plug-in" möjlighet i jwt_extended

Definiera egen funktion som kollar mot egen deny list

# Token kollas nu automatiskt mot egen deny list

```
@app.route('/protected', methods=['GET’])
@jwt_required()
def protected():
    return jsonify({'hello': 'world’})
```

Kommer kolla token även mot deny list

- Läs mer på:
  - https://flask-jwt-extended.readthedocs.io/en/latest/blocklist_and_token_revoking/

# OAuth 2.0

Open Authorization protocol

# OAuth 2.0: protokoll för inloggning

- Vilka parter involverade
- Vilka steg ska genomföras
- Vad ska skickas i varje steg

- Bra länkar:
  - https://auth0.com/docs/tokens/idp
  - https://aaronparecki.com/oauth-2-simplified/

# Normalt tre parter

- Resurs-server (hanterar databas, etc.)
- Klienten (= ‘user agent’)
  - Android-delen av er app (nu fejkad genom requests-anrop)
- Resurs-ägaren (användaren som äger sina data i databasen)

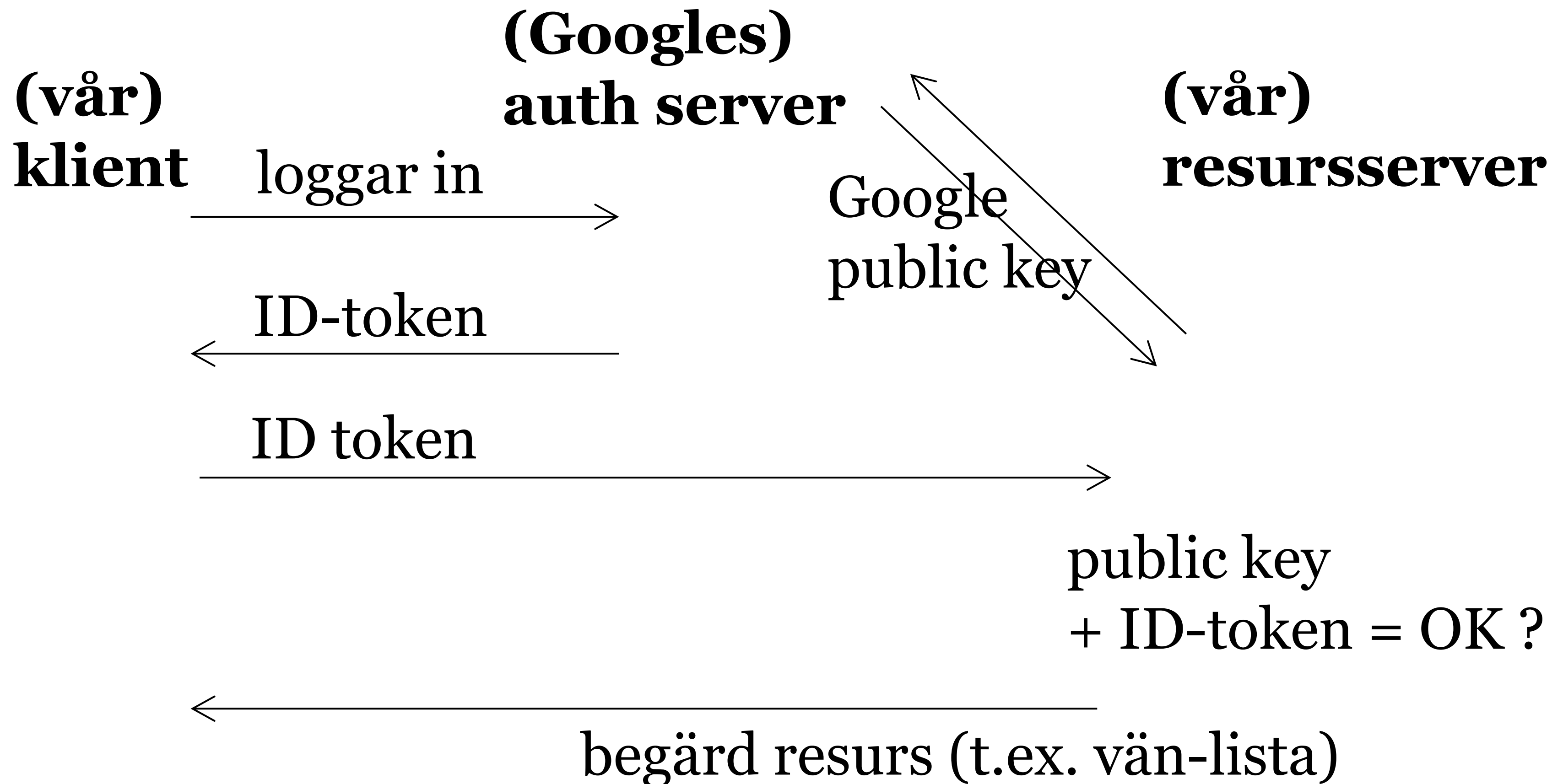
# Google Sign-In for Android

-- med egen backend server

# Tredjepartsinloggning (t.ex. Google signin)

- (Användare
  - Loggar in)
- Klienten (Android-delen i appen)
  - Använder Google-bibliotek för att visa inlogg-skärm i Android
- Auth-server (t.ex. Google ID)
  - Låter användaren logga in, returnerar ID-token
- Resurs-server (backend i appen)
  - Kontrollerar ID-token

# Autentiseringsflödet



# Googles auth-server

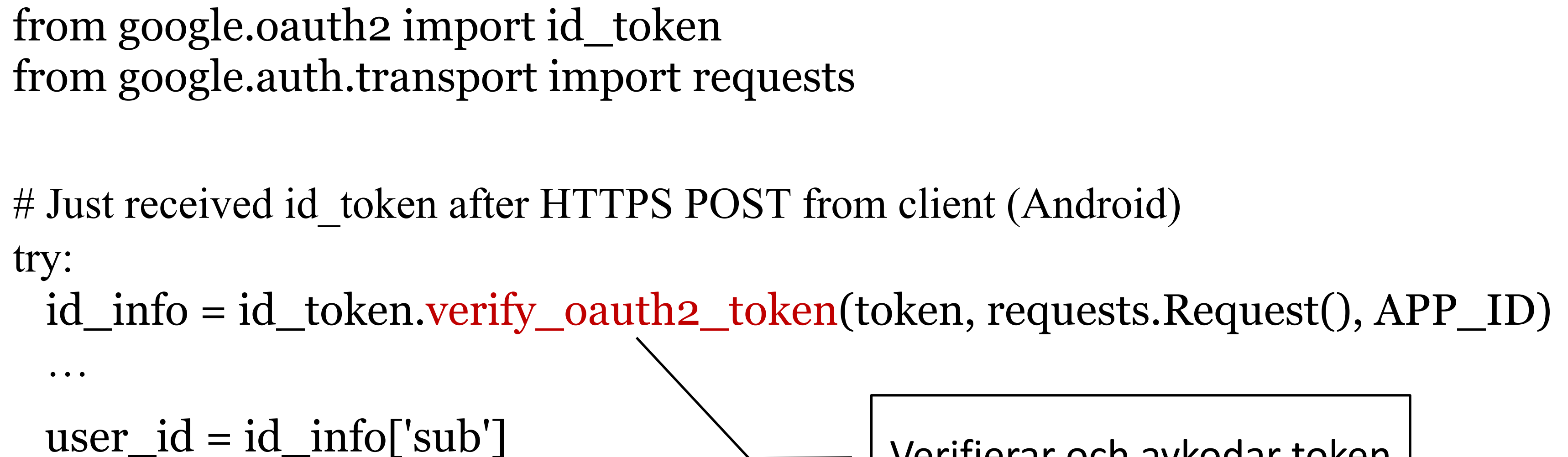
- Hanterar två säkerhetsrisker
  - Er app (backend)
  - Klienten (Android-delen)
- Båda måste identitetskollas
  - Appen reggas och får en app secret
  - Klienten loggar in, får ID-token
    - Användaren ombeds auktorisera att appen får tillgång till Googles användardata

# Koll av token på server-sidan

- Använd Googles public key för att verifiera integriteten av ID-token
  - Googles python bibliotek: https://developers.google.com/api-client-library/python/start/installation

# På server-sidan

```
>> pip3 install google-auth
```



# Claims i en Google ID token

```
{// These six fields are included in all Google ID Tokens.
"iss": "https://accounts.google.com",
"sub": "110169484474386276334",
"azp": "1008719970978-hb24n2dstb40o45d4feuo2ukqmcc6381.apps.googleusercontent.com",
"aud": "1008719970978-hb24n2dstb40o45d4feuo2ukqmcc6381.apps.googleusercontent.com",
"iat": "1433978353", "exp": "1433981953",

// These seven fields are only included when the user has granted the "profile" and
// "email" OAuth scopes to the application.
"email": "testuser@gmail.com",
"email_verified": "true",
"name" : "Test User",
"picture": "https://lh4.googleusercontent.com/-
kYgzyAWpZzJ/ABCDEFGHI/AAAJKLMNOP/tIXL9Ir44LE/s99-c/photo.jpg",
"given_name": "Test",
"family_name": "User",
"locale": "en"}
```

# Läs mer om Google Sign-in for web-backend

- Google Sign-in, web-backend:

https://developers.google.com/identity/sign-in/web/backend-auth

- Googles auth-bibliotek:

https://google-auth.readthedocs.io/en/latest/

- Googles API-client bibliotek:

https://developers.google.com/api-client-library/python/start/installation

rita.kovordanyi@liu.se

www.liu.se

LINKÖPINGS UNIVERSITET