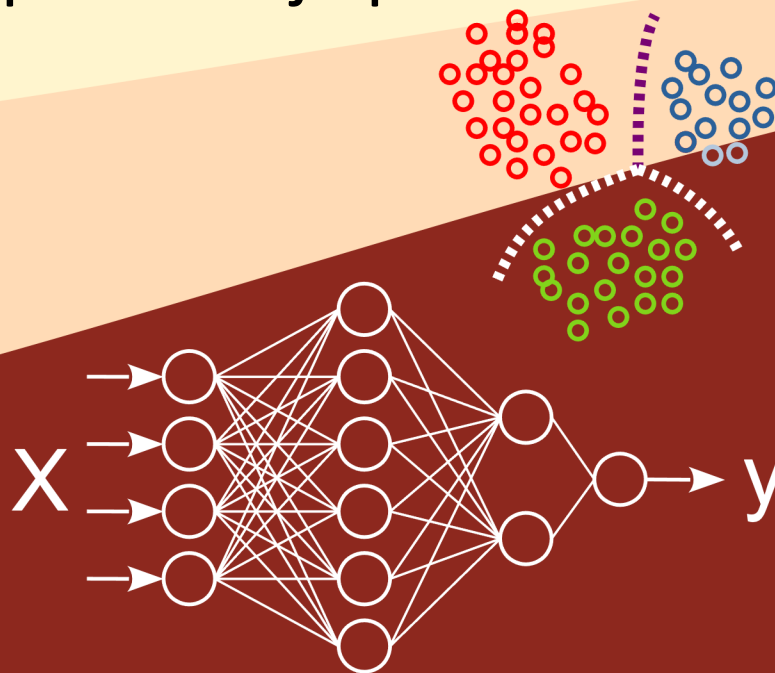
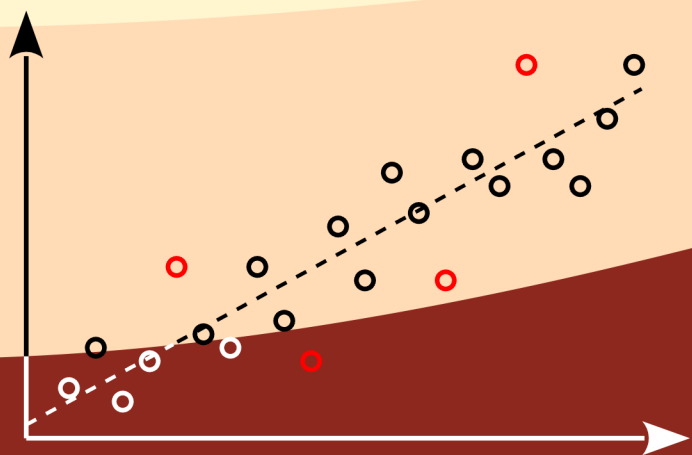


Arhitektura i Razvoj Inteligentnih Sustava

Tjedan 7: Servisna arhitektura, Sigurnost,
Inteligentna optimizacija procesa



Creative Commons



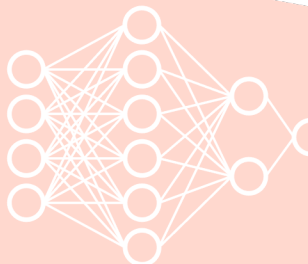
- slobodno smijete:

- dijeliti — umnožavati, distribuirati i javnosti priopćavati djelo
- prerađivati djelo



- pod sljedećim uvjetima:

- imenovanje: morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
- nekomercijalno: ovo djelo ne smijete koristiti u komercijalne svrhe.
- dijeli pod istim uvjetima: ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, prerađivanje možete distribuirati samo pod licencom koja je ista ili slična ovoj.



U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.

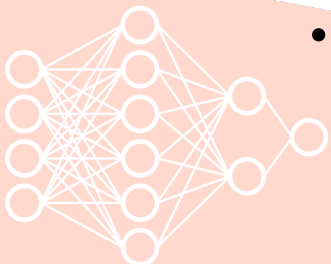
Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.

Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.

Tekst licence preuzet je s <http://creativecommons.org/>

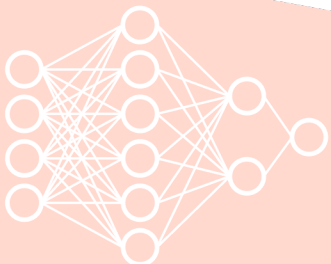
Posluživanje modela

- Jednom naučeni model želimo izložiti našoj organizaciji kako bi ga mogla pozivati i koristiti
- Osnovni način je izlaganje servisa (usluge)
 - Servis predstavlja omotač oko modela kojim simuliramo fazu testiranja modela jer nema ažuriranja hiperparametara
 - Tako izložen servis treba tehnologijom odgovarati okolini u kojoj se nalazi
 - Servis unutar sebe treba odraditi transformaciju modela
 - ML python moduli često imaju svoje tipove podataka koji nisu općenito i široko korišteni i priznati – u svrhu omogućavanja prijenosa podataka i parametara kroz vlastite strukture (gradijenti recimo)
 - Osim transformacije, možda treba odraditi i obogaćivanje podataka (*enrichment*)
 - Samo obogaćivanje podataka je kompleksan proces koji ponekad zahtijeva pozive prema vanjskim sustavima ili dohvat podataka iz baze podataka



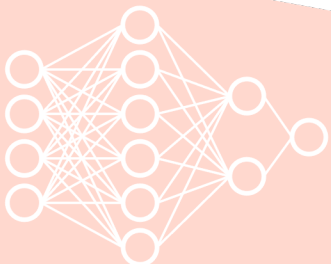
Posluživanje modela

- Takva transformacija i obogaćivanje podataka često je proces sam za sebe koji se često u kontekstu ML inženjerstva naziva i servisni cjevovod (*service pipeline*)
 - Servisni cjevovod može biti dio omotača, ali i ne mora
 - Servisni cjevovod se može implementirati kao kratki proces u *workflow engine-u* – recimo u Apache Airflow-u
-
- Kad govorimo o tehnologiji samog sučelja servisa, imamo dvije mogućnosti:
 - Standardni web servisi: SOAP / XML
 - Mikroservisi: REST / JSON



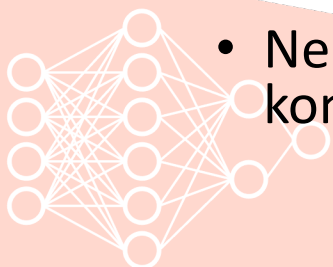
Standardni (*legacy*) servisi

- Standardna servisna arhitektura (SOA)
 - Koristi se HTTP protokol
 - Imamo jedan *endpoint* koji predstavlja servis
 - *Payload* je XML sa standardiziranom SOAP shemom
 - Sadrži *header*, *body* i *fault* elemente
 - Metode servisa adresiraju se kroz SOAP *payload*
 - Podaci / objekti predaju se kroz tijelo SOAP *payload-a*
 - Obično je to XML sastavljen prema nekoj konkretnoj shemi (XSD)
 - Taj XML je *strong typed* jer treba slijediti shemu – može se provjeriti prema toj shemi
 - Negativne strane pristupa – veliki *overhead*
 - U kodu koji se generira iz gomile opisnika
 - U provjeri ispravnosti XML-a
 - U veličini *payload-a*
 - U serijalizaciji / deserijalizaciji



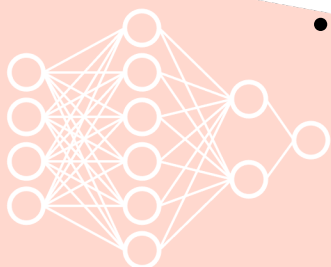
RESTful servisi

- REST = *REpresentational State Transfer*
- Također koriste HTTP – u većoj mjeri nego standardni servisi
 - Adresiranje ide kroz URL, koristi se HTTP metoda, status (200 OK), odgovor
 - Podržava razne *payload-e* – iako je text/json najzastupljeniji
 - JSON je neformalni – iako postoje načini kako provjeriti JSON *payload*
 - Može se koristiti i XML – čime se ponešto približavamo standardnim servisima i mogućnostima provjere koje nudi XML
 - JSON je blizak python programskom modelu – vidimo recimo *dictionary*
- Neki osnovni nedostatci
 - Nema perzistencije poziva
 - Nema garancije dostavljanja poziva – timeout nije nužni signal da nešto nije dostavljeno
 - Nema transakcije, nema atomarnosti – sve uhvaćene ili neuhvaćene greške se moraju kompenzirati – suprotne aktivnosti (*insert -> remove*)



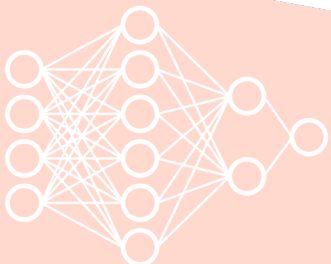
Inteligentni servis

- Omotamo model u kod koji nam predstavlja servisni cjevovod
- Sučelje tog omotača stavimo kao REST / JSON
- Zadaća omotača je
 - Transformacija i obogaćivanje podataka
 - Učitavanje modela i objekta za skaliranje
 - Sjetimo se, skaliranje je napravljeno prema određenoj statističkoj slici, pa je time unutarnje stanje objekta za skaliranje (recimo *StandardScaler*) bitno
 - Nakon učenja modela taj model i objekt za skaliranje spremimo
 - U mlflow – koji nam omogućava da kasnije to opet pročitamo
 - U *cloud* storage – recimo AWS, GS, minIO
 - Prilikom stvaranja omotača (u konstruktoru) ili prilikom prvog poziva mikroservisa – učitamo model i objekt za skaliranje u radnu memoriju
 - Svaki poziv je mikro-test za taj učitani model



RESTful servisi u pythonu - *flask*

- Za definiranje RESTful servisa u pythonu koristimo modul *flask*
- U development okolini ne trebamo puno paziti na arhitekturu
- U produkcijskoj okolini želimo imati pravi slijed *reverse proxy-a*
 - Tu možemo koristiti *unicorn* i *nginx*
- *flask* omogućava i funkcionalni i objektni pristup
- Kod objektnog pristupa imamo klasu koja se zatim instancira
 - Za svaku HTTP metodu definiramo posebnu metodu koja se poziva
 - Klasa / objekt ima svoj specifični endpoint koji definiramo *flask* anotacijama



flask – jednostavni primjer

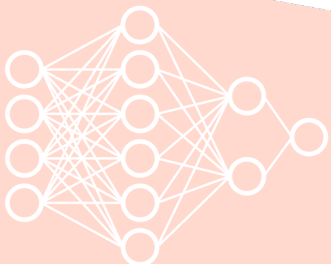
- Definiramo naziv aplikacije
- Definiramo URL za naš REST *handler*
- Metoda koju koristimo je GET
 - Vraćamo jednostavan *plain/text* s pozdravnom porukom
- RESTful servis se veže (*bind*) na <http://localhost:6789> čime će URL mikroservisa biti
 - <http://localhost:6789/test>

```
from flask import Flask
from flask_restx import Resource, Api

app = Flask("testapp")
api = Api(app)

@api.route('/test')
class TestApp(Resource):
    def get(self):
        return 'Hello, World!'

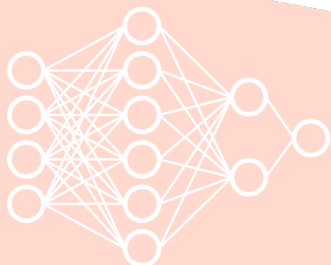
if __name__ == "__main__":
    app.run(host="localhost", debug=False, port=6789)
```



Inteligentni servis u *flask-u*

- Prethodno naučimo MLP sa *iris* skupom podataka – sjetimo se prethodnih laboratorijskih vježbi
 - Spremimo model i objekt za skaliranje u *pickle* (ili *joblib*) formatu u neki repozitorij – bilo mlflow ili minIO recimo
 - Sjetimo se, mlflow može koristiti minIO kao repozitorij artefakata
- Napravimo *handler* za mikroservis
 - Definiramo JSON *payload* za taj mikroservis

```
{"sepal_length":4.8,  
 "sepal_width":3.4,  
 "petal_length":1.6,  
 "petal_width":0.2}
```

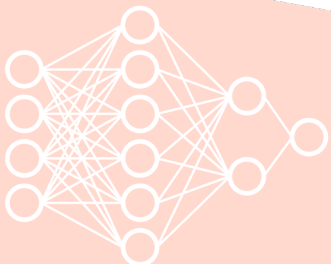


```
<imports>  
  
app = Flask("IrisModel")  
api = Api(app)  
  
iris_data = api.model(  
    'Iris Data', {  
        "sepal_length": fields.Float(description = "Sepal Length", required = True),  
        "sepal_width": fields.Float(description = "Sepal Width", required = True),  
        "petal_length": fields.Float(description = "Petal Length", required = True),  
        "petal_width": fields.Float(description = "Petal Width", required = True)  
    }  
)  
  
@api.route("/iris")  
@api.expect(iris_data)  
class IrisMSHandler(Resource):  
    def _load_model(self):  
        <učitavanje modela i objekta za skaliranje u self.mlp i self.scaler>  
  
    def post(self):  
        if not hasattr(self, "mlp"): self._load_model()  
        idata = request.get_json()  
        pdidata = pd.DataFrame(idata, index=[0])  
        pdidata_s = self.scaler.transform(pdidata)  
        pdidata_s = pd.DataFrame(pdidata_s)  
        predict = self.mlp.predict(pdidata_s)  
        return int(predict[0])  
  
if __name__ == "__main__":  
    app.run(host="localhost", debug=False, port=6789)
```

Inteligentni servis u *flask-u*

- Prethodno naučimo MLP sa *iris* skupom podataka – sjetimo se prethodnih laboratorijskih vježbi
 - Spremimo model i objekt za skaliranje u *pickle* (ili *joblib*) formatu u neki repozitorij – bilo mlflow ili minIO recimo
 - Sjetimo se, mlflow može koristiti minIO kao repozitorij artefakata
- Napravimo *handler* za mikroservis
 - Definiramo JSON *payload* za taj mikroservis

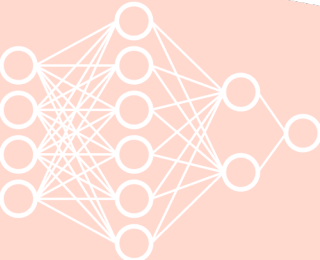
```
{"sepal_length":4.8,  
 "sepal_width":3.4,  
 "petal_length":1.6,  
 "petal_width":0.2}
```



```
<imports>  
  
app = Flask("IrisModel")  
api = Api(app)  
  
iris_data = api.model(  
    'Iris Data', {  
        "sepal_length": fields.Float(description = "Sepal Length", required = True),  
        "sepal_width": fields.Float(description = "Sepal Width", required = True),  
        "petal_length": fields.Float(description = "Petal Length", required = True),  
        "petal_width": fields.Float(description = "Petal Width", required = True)  
    }  
)  
  
@api.route("/iris")  
@api.expect(iris_data)  
class IrisMSHandler(Resource):  
    def _load_model(self):  
        <učitavanje modela i objekta za skaliranje u self.mlp i self.scaler>  
  
    def post(self):  
        if not hasattr(self, "mlp"): self._load_model()  
        idata = request.get_json()  
        pdidata = pd.DataFrame(idata, index=[0])  
        pdidata_s = self.scaler.transform(pdidata)  
        pdidata_s = pd.DataFrame(pdidata_s)  
        predict = self.mlp.predict(pdidata_s)  
        return int(predict[0])  
  
if __name__ == "__main__":  
    app.run(host="localhost", debug=False, port=6789)
```

Inteligentni servis u *flask-u*

- Svaki se mikroservis može dokumentirati, što završava u *Swagger* URL-ovima
 - <http://localhost:6789/>
 - <http://localhost:6789/swagger.json>



Models	
Iris Data ▾ {	
sepal_length*	number Sepal Length
sepal_width*	number Sepal Width
petal_length*	number Petal Length
petal_width*	number Petal Width
}	

```
<imports>

app = Flask("IrisModel")
api = Api(app)

iris_data = api.model(
    'Iris Data', {
        "sepal_length": fields.Float(description = "Sepal Length", required = True),
        "sepal_width": fields.Float(description = "Sepal Width", required = True),
        "petal_length": fields.Float(description = "Petal Length", required = True),
        "petal_width": fields.Float(description = "Petal Width", required = True)
    }
)

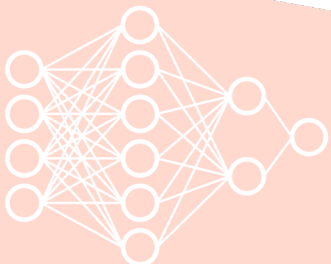
@api.route("/iris")
@api.expect(iris_data)
class IrisMSHandler(Resource):
    def _load_model(self):
        <učitavanje modela i objekta za skaliranje u self.mlp i self.scaler>

    def post(self):
        if not hasattr(self, "mlp"): self._load_model()
        idata = request.get_json()
        pdidata = pd.DataFrame(idata, index=[0])
        pdidata_s = self.scaler.transform(pdidata)
        pdidata_s = pd.DataFrame(pdidata_s)
        predict = self.mlp.predict(pdidata_s)
        return int(predict[0])

if __name__ == "__main__":
    app.run(host="localhost", debug=False, port=6789)
```

Inteligentni servis u *flask-u*

- Primijetimo neke detalje
 - JSON *payload* za mikroservis
 - Pretvaramo ga prvo u objekt sličan *dictionary-u*
 - Zatim ga konvertiramo u pandas *DataFrame*
 - Nakon toga koristimo objekt za skaliranje kako bismo ga ispravno skalirali
 - Pozivamo *predict* metodu MLP klasifikatora
 - Vraćamo labelu koju smo dobili od klasifikatora



```
<imports>

app = Flask("IrisModel")
api = Api(app)

iris_data = api.model(
    'Iris Data', {
        "sepal_length": fields.Float(description = "Sepal Length", required = True),
        "sepal_width": fields.Float(description = "Sepal Width", required = True),
        "petal_length": fields.Float(description = "Petal Length", required = True),
        "petal_width": fields.Float(description = "Petal Width", required = True)
    }
)

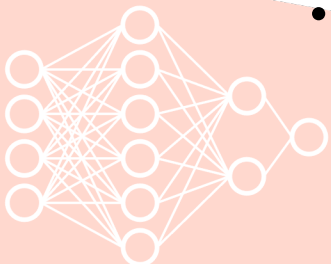
@api.route("/iris")
@api.expect(iris_data)
class IrisMSHandler(Resource):
    def _load_model(self):
        <učitavanje modela i objekta za skaliranje u self.mlp i self.scaler>

    def post(self):
        if not hasattr(self, "mlp"): self._load_model()
        idata = request.get_json()
        pdidata = pd.DataFrame(idata, index=[0])
        pdidata_s = self.scaler.transform(pdidata)
        pdidata_s = pd.DataFrame(pdidata_s)
        predict = self.mlp.predict(pdidata_s)
        return int(predict[0])

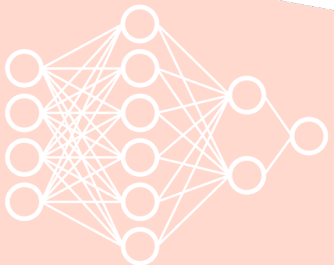
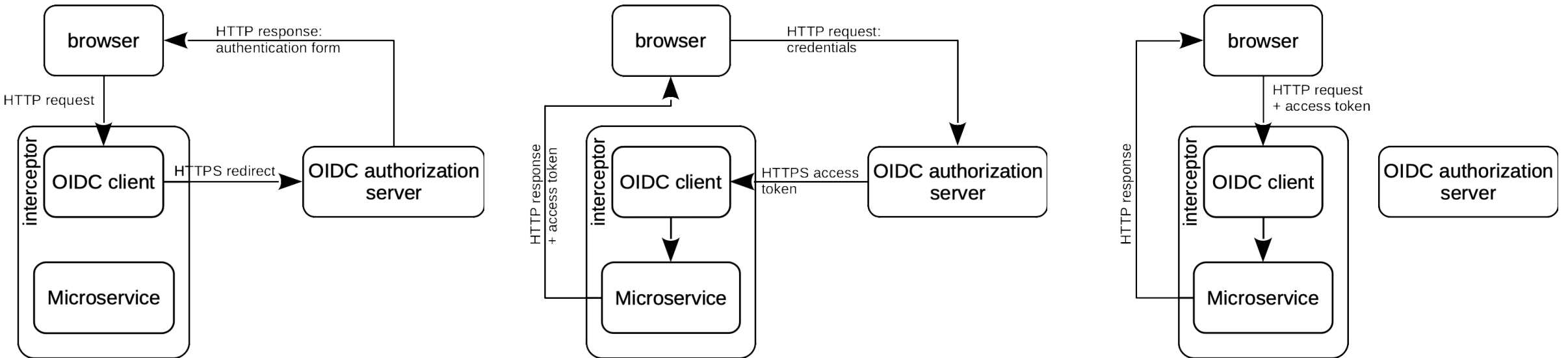
if __name__ == "__main__":
    app.run(host="localhost", debug=False, port=6789)
```

Sigurnost mikroservisa

- Transportna sigurnost
 - SSL / HTTPS – obostrano korištenjem serverskog i klijentskog certifikata
 - Jednostrano – uspostava tunela sa serverom – barem znamo da je server taj za koji se izdaje
- Što je s autentifikacijom i autorizacijom nad mikroservisima?
 - Moderni protokoli – *OpenID Connect* (OIDC) protokol koji radi nad *Open Authentication 2.0* okvirom (<https://openid.net/connect/>)
 - Podržava više različitih autorizacijskih servera – npr. *keycloak* (RedHat)
 - Više načina (tokova) za autentifikaciju korisnika
 - Idealno za mikroservise i tanke klijente (recimo *ReactJS*)
 - Sve što u osnovi koristi HTTP

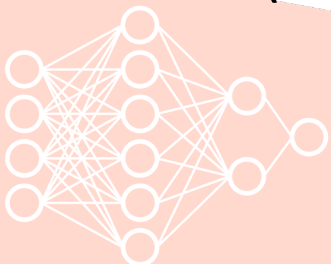


OIDC standardni tok (*standard flow*)



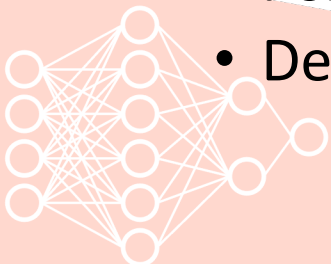
OIDC

- Koncept se temelji na *access* i *refresh* tokenima (<https://jwt.io>)
 - Svaki token ima svoje trajanje
 - *access* token sadrži podatke o korisniku (*user*)
 - To ovisi o opcijama za mapiranje u autorizacijskom serveru
 - Osobni podaci, uloge – ovo se sve može definirati
 - *refresh* token traje dulje i koristi se za obnavljanje *access* tokena
 - Server vjeruje *access* tokenu i daje pristup u ovisnosti o podacima u tom tokenu
 - Tokeni jednostavno enkriptirani (BASE64) i potpisani
 - Svaki se token može provjeriti na autorizacijskom serveru – posebni tokovi (kao npr. *bearer* autentifikacija – *direct access grant*)



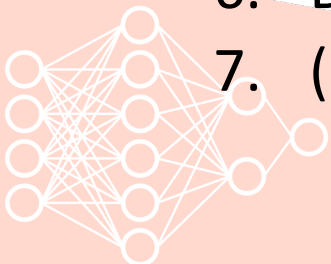
OIDC *bearer* autentifikacija

- Klijent (*client*, nije isto što i *user*) sam kontaktira OIDC autorizacijski server i traži token – nema redirekcije – *direct access grant*
- Ovo potraživanje tokena može ići kroz SSL tunel
- Klijent se kroz SSL tunel autentificira autorizacijskom serveru
- Šalju se korisnički (*user*) podaci – dobivaju se tokeni
- Prilikom poziva mikroservisa, token se stavlja u HTTP zaglavlje
- Poslužitelj koji poslužuje mikroservis prvi puta kada uoči novi *access token*
 - Pozadinska provjera s autorizacijskim serverom
 - Definiranje prava nad mikroservisom



keycloak – OIDC server

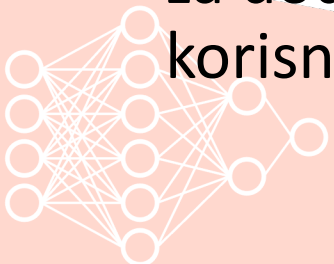
- <https://www.keycloak.org>
- Temeljen na Javi – moguće pokretanje na raznim aplikacijskim serverima (!?) – Quarkus, WildFly
- Traži nešto laganog postavljanja
 1. Stvorimo sigurnosnu domenu (*security realm*)
 2. Stvorimo klijenta – *standard flow*
 3. Podesimo autentifikaciju za klijenta – *client id / client secret*
 4. Definiramo ulogu (*role*) na klijentu
 5. Definiramo grupu korisnika i dodamo joj ulogu
 6. Definiramo korisnika (i lozinku), te ga pridodamo grupi
 7. (opcija) Podesimo mapiranje za korisničke podatke



OIDC klijentska konfiguracija

- Definiramo listu RESTful *endpoint-a*
- Definiramo identifikaciju klijenta
- U slučaju da se klijent autentificira kod autorizacijskog servera dodamo i *client secret* – time se i klijent predstavlja autorizacijskom serveru
- Redirect URIs – Lista URL-a na koje se vraćaju tokeni
- Ostalo su *endpoint-i* koje autorizacijski server daje za dodatne podatke o korisniku i tokenima

```
{
  "web": {
    "issuer": "http://localhost:8080/realms/aris",
    "auth_uri": "http://localhost:8080/realms/aris/protocol/openid-connect/auth",
    "client_id": "flask",
    "client_secret": "TwQy3aWWH4NKVMiAb5UFaKJcAUVbBSs0",
    "redirect_uris": [
      "http://localhost:6789/*"
    ],
    "userinfo_uri": "http://localhost:8080/realms/aris/protocol/openid-connect/userinfo",
    "token_uri": "http://localhost:8080/realms/aris/protocol/openid-connect/token",
    "token_introspection_uri": "http://localhost:8080/realms/aris/protocol/openid-connect/token/introspect"
  }
}
```



flask OIDC

- Koristimo modul *flask_oidc* u pythonu
- U konfiguraciji pokazujemo na json (prethodni *slide*) s kojim definiramo OIDC detalje
- Dodatno definiramo sigurnosnu domenu i područja (*client scope*) za našeg klijenta
- Dodajemo anotacije na metode koje želimo zaštititi – recimo na ovoj smo metodi tražili samo da korisnik bude autentificiran
- Provjerite [flask-oidc](#) dokumentaciju

```
from flask import Flask
from flask_restx import Resource, Api
from flask_oidc import OpenIDConnect

app = Flask("securetest")
app.config.update({
    'SECRET_KEY': 'test',
    'OIDC_CLIENT_SECRETS': 'client_secrets.json',
    'OIDC_ID_TOKEN_COOKIE_SECURE': False,
    'OIDC_REQUIRE_VERIFIED_EMAIL': False,
    'OIDC_USER_INFO_ENABLED': True,
    'OIDC_OPENID_REALM': 'aris',
    'OIDC_SCOPES': ['openid', 'profile'],
    'OIDC_INTROSPECTION_AUTH_METHOD': 'client_secret_post'
})
oidc = OpenIDConnect(app)
api = Api(app)

@api.route('/sectest')
class SecureTestApp(Resource):
    @oidc.require_login
    def get(self):
        return 'Hello, ' + oidc.user_getfield('name')

if __name__ == "__main__":
    app.run(host="localhost", debug=False, port=6789)
```



flask OIDC

- Nakon što upišemo URL <http://localhost:6789/sectest> u browser, dobivamo redirekciju na keycloak formu za login
- Upišemo korisničko ime i lozinku, nakon čega keycloak redirekcijom dostavlja tokene našem mikroservisu
- Tokeni se vrate u browser
- Sljedeći poziv mikroservisa više ne zahtijeva korisničke podatke – sve do dok vrijedi *access token*

```
from flask import Flask
from flask_restx import Resource, Api
from flask_oidc import OpenIDConnect

app = Flask("securetest")
app.config.update({
    'SECRET_KEY': 'test',
    'OIDC_CLIENT_SECRETS': 'client_secrets.json',
    'OIDC_ID_TOKEN_COOKIE_SECURE': False,
    'OIDC_REQUIRE_VERIFIED_EMAIL': False,
    'OIDC_USER_INFO_ENABLED': True,
    'OIDC_OPENID_REALM': 'aris',
    'OIDC_SCOPES': ['openid', 'profile'],
    'OIDC_INTROSPECTION_AUTH_METHOD': 'client_secret_post'
})
oidc = OpenIDConnect(app)
api = Api(app)

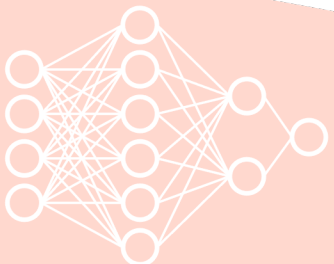
@api.route('/sectest')
class SecureTestApp(Resource):
    @oidc.require_login
    def get(self):
        return 'Hello, ' + oidc.user_getfield('name')

if __name__ == "__main__":
    app.run(host="localhost", debug=False, port=6789)
```



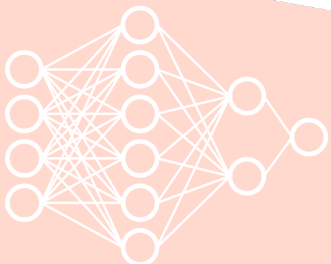
OIDC Single-Sign On (SSO)

- Dok je standardni tok uglavnom za osiguranje tankih klijenata (*ReactJS*), *bearer* autentifikacija (*direct access grant*) se može upotrijebiti za uspostavu SSO
- Jedan sustav (ulazni) dohvat tokene od autorizacijskog servera
- Tokeni se zatim propagiraju kroz integracije
 - Svaki sustav odlučuje da li vjeruje tokenu i da li ga želi provjeriti
- U trenutku kada *access* token prestane vrijediti, ulazni sustav koristi *refresh* token da dohvati novi *access* token



Sigurnost inteligentnog servisa

- Pristup samom REST servisu modela može se ograničiti mrežno
 - Definiramo unutarnje IP adrese koje smiju koristiti REST servis
- Pristup REST sučelju servisnog cjevovoda ima smisla ograničiti na korisnike određene sigurnosne domene i/ili korisnike koji su određenoj ulozi
- Uspostava SSO-a je dobar način osiguranja servisnih cjevovoda
- O mogućim napadima na model ili postupak učenja i efektima istog govorit ćemo u narednim predavanjima



Inteligentna optimizacija procesa

- Cilj svake organizacije je optimizirati svoje poslovne procese na način da se maksimiziraju ključni pokazatelji performansi procesa (KPI)
- Postoje točke u poslovnom procesu koje predstavljaju
 - Optimizacijske zadatke – recimo optimizacija raspoređivanja
 - Točke odluka
 - Modeli koji upravljaju poslovnim procesom
- Cjevovod za učenje se pokreće kad KPI-evi padnu ispod određene razine

