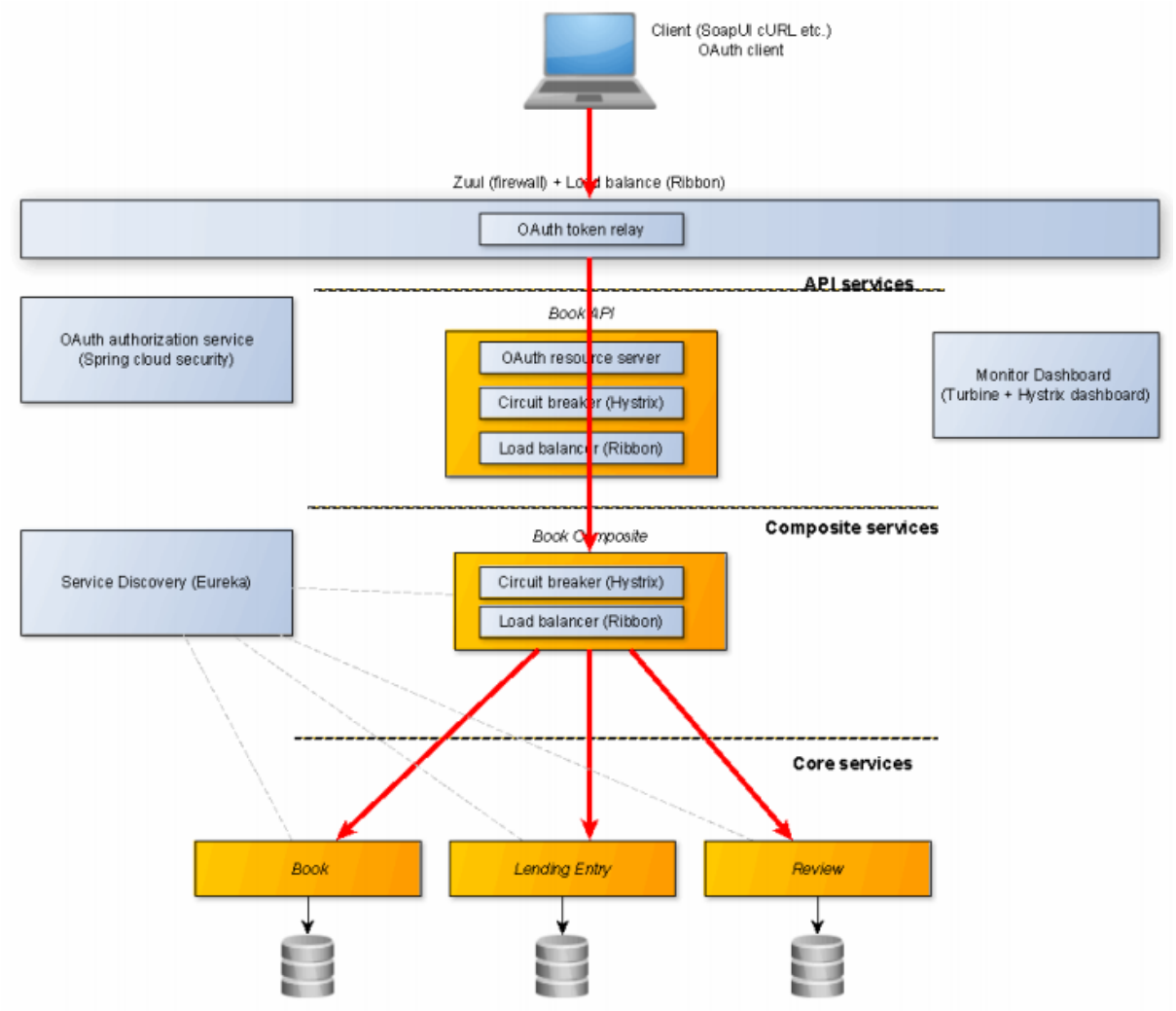
Laboratorium mikroserwisy

Na tym laboratorium dowiesz się czym jest mikroserwis, do czego służy, kiedy jest stosowany oraz jak go stworzyć i połączyć z bazą danych w 3 bardzo popularnych technologiach springBoot, flask, express.

Mikroserwisy to stosunkowo nowe podejście do tworzenia architektury aplikacji webowej. Klasycznie aplikacje stanowiła monolit (pojedynczy program, który robił wszystko), podejście mikroserwisowe polega na rozbiciu monolitycznej aplikacji na wiele mniejszych części, gdzie każda część powinna zajmować się jedynie swoją funkcjonalnością.



Przykład architektury mikroserwisowej, z netflix oss.

Prowadzi to do stworzenia aplikacji rozproszonej, która można znacznie lepiej skalować i rozwijać natomiast wiąże się to z dodatkowym narzutem na początkowym etapie prac oraz aplikacji pomocniczych jak Discovery service jak i zwiększeniem skomplikowania aplikacji. Podejście stosowane jest głównie przez duże firmy o znaczącym ruchu w systemie w szczególności gdy ruch nie jest równomiernie rozłożony.

1. Dziś zrealizujemy proste serwery w 3 popularnych technologiach, wszystkie serwery będą realizowały tę samą funkcjonalność, mianowicie endpointy get i post oraz połączenie i obsługa przykładowej bazy danych.
2. SpringBoot spring cloud jak zacząć projekt https://start.spring.io/ zaznaczamy spring web z prawej strony, zmieniamy nazwę, paczkę i pobieramy wygenerowany szkielet.
    a. Otwieramy z użyciem ulubionego ide. I testujemy czy się uruchamia.
    b. Przydadzą się nam takie dependencje:

```
implementation
'org.springframework.boot:spring-boot-starter-web'
implementation
'org.springframework.boot:spring-boot-starter-data-jpa'
runtimeOnly 'com.h2database:h2'
testImplementation
'org.springframework.boot:spring-boot-starter-test'
```

    c. Jeżeli wszystko działa dodajmy sobie pierwszy kontroler.
        i. Tworzymy klasę hello w pakiecie controller
        ii. w klasie hello dodajemy publiczną funkcję zwracającą string i przyjmująca jeden argument name.
        iii. Dodajemy odpowiednie adnotacje `@RestController` do klasy, `@GetMapping()` do metody i `@RequestParam()` do parametru name.
        iv. Funkcja powinna zwrócić cześć + $name.
        v. Sprawdzamy, czy działa. No ale jak? http://localhost:8080/hello (ps.jeżeli nie działa to czy włączyłeś serwer ?)
    d. Zmieńmy stringa na Json, jak to zrobić? ręcznie? nie lepiej użyć POJO a jeszcze lepiej record.
    e. Zapakujmy odpowiedź, tak by dodać nagłówki (`ResponseEntity`).
    f. Dodajmy obsługę bazy danych spring data + jpa.
        i. w tym celu potrzebne będzie nam model danych, repozytorium, i serwis i minimalna konfiguracja.
        ii. model danych:

```
@Entity
public class Person {


    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    public Long id;
    public String name;
    public String surname;
    public String job;
      +  gettery, settery i konstruktory
```

        iii. application yaml

```
spring:
 datasource:
    url: jdbc:h2:file:./data/demo
    username: sa
```

```
    password: password
    driverClassName: org.h2.Driver
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    defer-datasource-initialization: true
    hibernate:
      ddl-auto: create-drop
```

iv. potrzebne będzie nam repozytorium

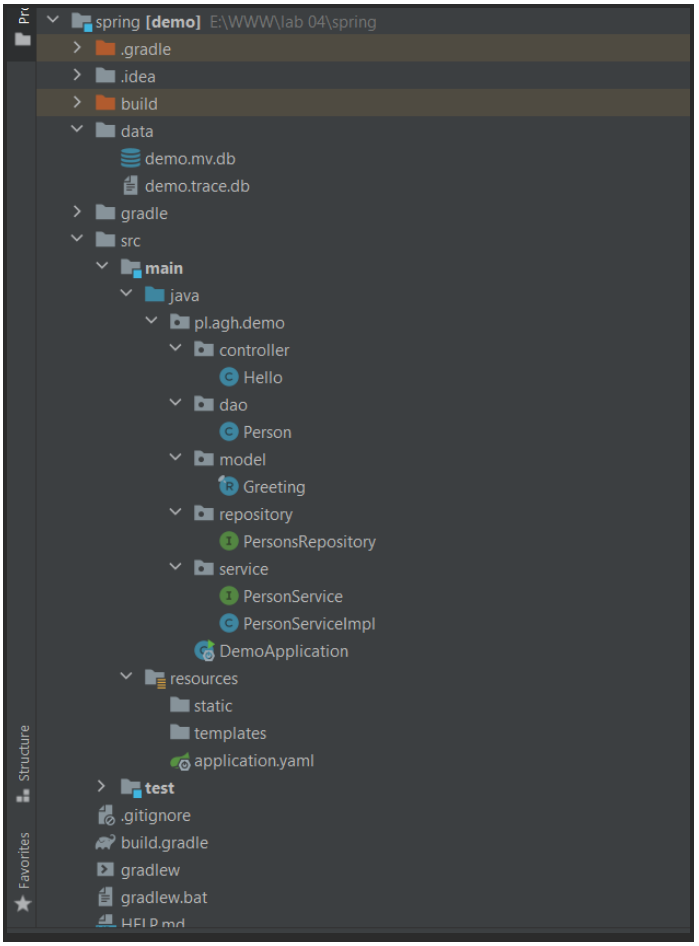
1. `PersonsRepository rozszerzające CrudRepository`
2. Repozytorium to interface, z użyciem którego bez żadnej implementacji możemy, uzyskać dostęp do danych. korzysta się z konwencji np. by pobrać wszystkie osoby z bazy danych findAll(), normalnie zwróci to nam iterator, ale możemy zmienić go na tym etapie na listę `List<Person> findAll();`
3. Stwórzmy sobie serwis, który będzie implementował taki interfejs

```
public interface PersonService {
    public List<Person> getPersons();
    public Person getPerson(String surname);
    public Person create(Person person);
    public Person getPerson(int id);
}
```

4. potrzebna nam implementacja naszego serwisu jego głównym zadaniem jest korzystanie z repozytorium i zwracanie danych.
   a. serwisy są adnotowane `@Service`
   b. Repozytorium tworzymy w serwisie na zasadzie “automagi”
   ```
   @Autowired
   private PersonsRepository personsRepository;
   ```
   c. podobnie w kontrolerze tworzymy serwis.
5. pozostało nam dodać jeszcze 3 endpointy
   a. `GET /person`
   b. `GET /person/{id}`
   c. post`/create` żeby dodać nową osobę.
6. Przetestujmy czy działa.
7. dodajmy jakieś dane do bazy w celach testowych:

```
@Bean
public CommandLineRunner demo(PersonsRepository
repository) {
  return (args) -> {
      // save a few customers
      repository.save(new Person("John", "Doe","IT"));
      repository.save(new Person("John",
"Smith","tester"));
      // fetch all customers
      log.info("Customers found with findAll():");
  log.info("-------------------------------");
```

```
        repository.findAll().forEach(customer -> {
            log.info(customer.toString());
        });
};
}
```



3. Flask
    a. Tworzymy sobie nowy projekt Python(jeżeli jest taka opcja to flask)
    b. endpointy podobnie jak poprzednio to funkcje z odpowiednią adnotacją tym razem `@app.route('/', methods=['GET'])` GET jest wartością domyślną i nie jest wymagane.

```
from flask import Flask, jsonify

app = Flask(__name__)

#tu umieść endpointy

if __name__ == '__main__':
    app.run()
```

c. Żeby pobrać wartość z query:
   i. path `@app.route('/<nazwa>')`
   ii. `def some_func(nazwa):`
   iii. `query:`
   iv. `nazwa = request.args["nazwa"]`

d. Jeżeli chcemy zwrócić jsona używamy typowo `jsonify() najlepiej przekazać mu słownik, obiekty pythona mają .__dict__`

e. Flask korzysta z SQLAlchemy jako ORM podobnie jak w spring data potrzebne będzie nam model oraz konfiguracja:

```
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import DeclarativeBase, Mapped,
mapped_column

app.config["SQLALCHEMY_DATABASE_URI"] =
"sqlite:///demo.sqlite"
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

class Base(DeclarativeBase):
 pass

class Person(db.Model):
   id: Mapped[int] = mapped_column(db.Integer,
primary_key=True)
   name: Mapped[str] = mapped_column(db.String)
   surname: Mapped[str] = mapped_column(db.String)
   job: Mapped[str] = mapped_column(db.String)
```

f. Żebyśmy się nie zmachali przy serializowaniu/deserializowaniu danych z bazy użyjemy sobie marshmalow

```
from flask_marshmallow import Marshmallow
#po sqlalchemy
ma = Marshmallow(app)

class PersonSchema(ma.SQLAlchemyAutoSchema):
   class Meta:
       model = Person
```

g. stwórzmy sobie i wyczyśćmy baze oraz dodajmy jakąś osobe

```
with app.app_context():
   db.drop_all()
   db.create_all()
```

```
db.session.add(Person(name="ser",surname="serowy",job='it'
))
   db.session.commit()
```

h. a jak pobrać dane z bazy

```
#wszystko
db.session.execute(db.select(Person)).scalars()
#kiedy id ==id albo 404
db.get_or_404(Person, id)
```

i. jak użyć marshmalow żeby sobie pomuc
   i. `PersonSchema().dump(person) #pojedyńcza wartość`
   ii. `PersonSchema(many=True).dump(person) #wiele wartości`

j. jak użyć marshmalow do deserializacji
   i. `PersonSchema().load(jsonData)`
   ii. jak chcemy odrazu obiekt person to do PersonSchema dodajemy

```
@post_load
def make_user(self, data, **kwargs):
   return Person(**data)
```

4. Express
   a. minimalna aplikacja

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
   res.send('Hello World!')
})

app.listen(port, () => {
   console.log(`Example app listening on port ${port}`)
})
```

   b. oczywiście my sobie to wygenerujemy
      i. albo przez ide albo
      ii. npx express-generator
      npm install
      $env:DEBUG='myapp:*'; npm start

   c. routing:

```
   i. router.get('/',(req,res,next)=>{})
  ii. router.get('/:value',(req,res,next)=>{
```

iii.
```
        const value = req.params.value;
```
iv.
```
    })
```
v.
```
    router.post('/',(req,res,next)=>{})
```
vi.

d. routery są w osobnych plikach niż główna plik i są w nim do aplikacji dodawane.
```
app.use('/', indexRouter);
```
e. odpowiadamy na zapytania wywołując
```
res.status(code).json(body);
```

f. no dobra hello za nami jak ogarnąć baze? Na początek biblioteki:
```
"sequelize": "6.34.0",
"sqlite3": "^5.1.6"
```

i. sqlite wystarczy, jak ktoś jest masochistom albo kocha sql. `sequelize` to interfejs wyższego poziomu.

ii. Stwórzmy połączenie z bazą:
```
var sequelize = new Sequelize('test', 'root', '', {
    host: 'localhost',
    dialect: 'sqlite',
    operatorsAliases: false,
    storage: './data/database.sqlite'
});
```

iii. Tworzymy model danych:
```
var PersonSchema = sequelize.define("Person", {
    id: {type: DataTypes.INTEGER, autoIncrement: true,
primaryKey: true},
    name: DataTypes.STRING,
    surname: DataTypes.STRING,
    job: DataTypes.STRING
},{

});
```

iv. eksportujemy.
```
module.exports = {sequelize, PersonSchema};
```

v. to wszystko w osobnym pliku np db.js

vi. w celu przetestowania i utworzenia bazy(app.js)
```
var models = require("./db")

models.sequelize.sync().then(function() {
    console.log('connected to database')
}).catch(function(err) {
    console.log(err)
});
models.PersonSchema.create({'name': 'john',
'surname':'Doe', 'job':'IT'})
```

vii. no i tworzymy nowy router dla person

viii. funkcje kture się przydadzą:

```
1. Schema.findAll()
2. Schema.findByPk()
3. Schema.create()
```

NA 5.0.

Dokończyć operacje CRUD dla person we wszystkich serwerach

Create	INSERT	PUT / POST
Read (Retrieve)	SELECT	GET
Update	UPDATE	POST / PUT / PATCH
Delete (Destroy)	DELETE	DELETE