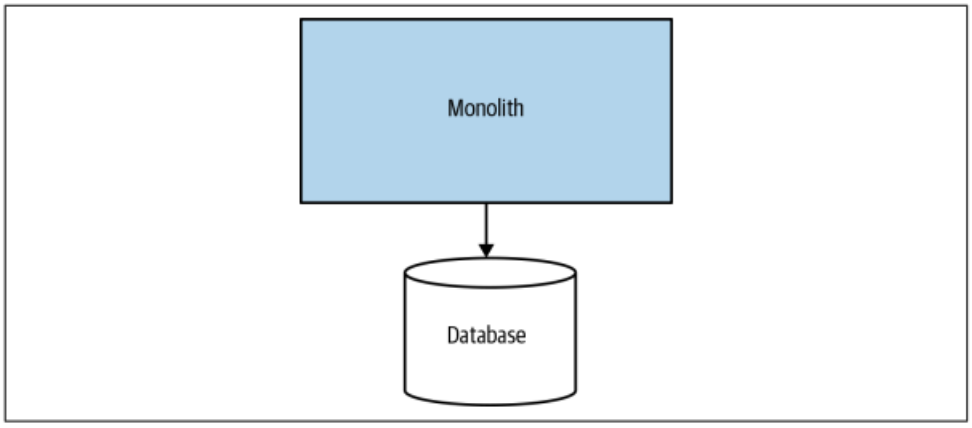
Laborator 6 SD: Microservicii Web cu Spring Boot în Kotlin

Introducere

Aplicații monolitice

Primele aplicații erau de dimensiuni mici, iar logica de prezentare era la un loc cu logica de business. Apoi, modelarea domeniului a devenit complexă, ceea ce a dat naștere mai multor modele de proiectare arhitecturală din categoria separării funcționalităților (Separation of Concerns).

În aplicațiile clasice monolit, toate funcționalitățile aplicației sunt grupate într-un singur pachet și executate ca un singur proces. Interfața cu utilizatorul, nivelul de accesare a datelor și nivelul de stocare a datelor sunt strâns cuplate.



Arhitectura unei aplicații monolit clasic

Avantaje:

- Sunt simplu de dezvoltat (toate instrumentele pentru dezvoltare suportă acest tip de aplicații)
- Sunt simplu de lansat (toate componentele fiind împachetate la un loc)
- Este o scalare ușoară a întregii aplicații

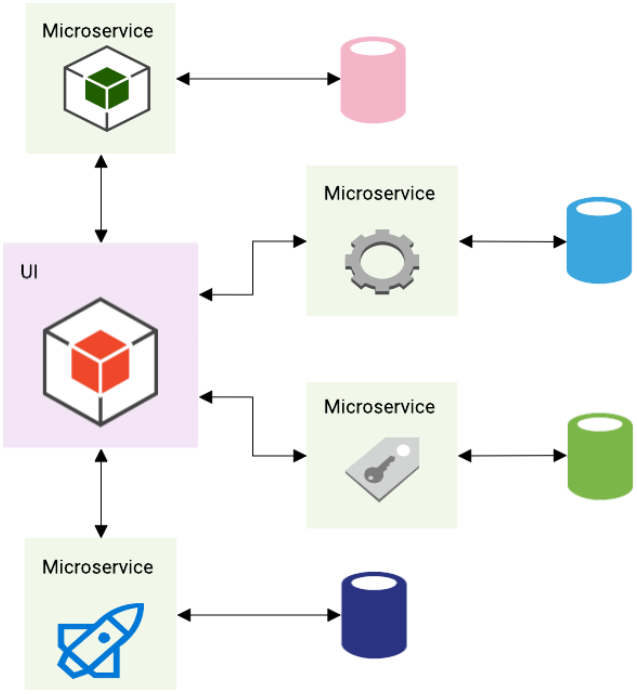
Dezavantaje:

- Sunt complexe și ca urmare este dificilă îmbunătățirea în timpul de viață.
- Adaptarea la noile tehnologii este destul de dificilă
- Sunt greu de integrat într-un proces continuu de dezvoltare CI/CD (Continuous Integration / Continuous Delivery)
- Pornirea aplicației durează destul de mult, din cauza faptului că toate componentele trebuie încărcate înainte de lansarea în execuție.
- Comportamentul eronat al unei componente va conduce la oprirea parțială sau totală a execuției

Aplicații cu microservicii

Termenul de microaplicații fost introdus în 2011 de către James Lewis de la ThoughtWorks care a început să studieze micro aplicațiile. Acesta era un model de proiectare care începuse deja să fie utilizat de unele companii care utilizau SOC. Ulterior, în 2012, în urma unor discuții la un congres de specialitate, s-a modificat în microservicii pentru a evita confuziile.

În cazul aplicațiilor web bazate pe microservicii acestea sunt formate dintr-o suită de servicii mici, executate independent, care comunică prin mecanisme simple (lightweight) bazate pe HTTP. Acestea au de multe ori un acces centralizat prin intermediul unui API.



Arhitectura decentralizată a microserviciilor

Avantaje:

- Microserviciile pot folosi cele mai noi tehnologii.
- Compozabilitatea este ridicată.
- Microserviciile independente pot fi scalate separat (nu e nevoie de scalarea întregului sistem).
- Defectarea unei componente nu va duce la căderea sistemului.
- Pentru dezvoltare se utilizează echipe mici care lucrează în paralel la dezvoltarea microserviciilor. Ca rezultat, timpul de dezvoltare se micșorează.
- Procesul de integrare/dezvoltare continuă este nativ.

Dezavantaje:

- Mentenanța codului de bază independent este foarte dificilă.
- Monitorizarea întregului sistem este o adevărată provocare, din cauza decentralizării.
- Are un cost (overhead) de performanță adițional din cauza latenței rețelei (network latency)

Pentru mai multe detalii, vezi:

- Building Microservices (2015) - Sam Newman
- Hands-on microservices with Kotlin (2018) - Juan Antonio Medina Iglesias

Principii generale pentru proiectarea cu microservicii

- **Modeled around business capabilities** - proiectarea software are o componentă de abstractizare, dezvoltatorii fiind obișnuiți să primească sarcini și să le implementeze, dar trebuie luat în considerare cum o să fie înțeleasă soluția, atât acum cât și în viitor. Se recomandă să se lucreze îndeaproape cu experții din domeniul respectiv.

- **Cuplare scăzută (Loosely couple)** - Nici un microserviciu nu există pe cont propriu, fiecare sistem având nevoie să interacționeze cu altele (alte microservicii), dar e nevoie ca această interacțiune să fie slab cuplată. De exemplu, dacă se proiectează un microserviciu care returnează numărul de oferte disponibile pentru un anumit client, este nevoie de o relație către clientul respectiv (customer ID), iar acesta ar fi nivelul maxim de cuplare acceptat.

- **Responsabilitate unică (single responsibility)** - Fiecare microserviciu are ca responsabilitate o singură parte din funcționalitatea aplicației, iar acea responsabilitate este încapsulată în interiorul lui.

- **Ascunderea implementării (Hiding implementation)** - Microserviciile au în general un contract (o interfață) clar și ușor de înțeles, care ascunde detaliile de implementare. Detaliile interne nu ar trebui expuse, nici implementarea tehnică, nici regulile de business care o conduc.

- **Izolare (Isolation)** - Un microserviciu trebuie izolat fizic și/sau logic de infrastructură care utilizează sistemul de care depinde (baza de date, server, etc). Astfel, se poate garanta că nimic extern nu poate afecta funcționalitatea aplicației, iar aplicația nu poate afecta ceva extern.

- **Instalare independentă (Independently deployable)** - Un microserviciu trebuie să poată fi instalat (deployed) în mod independent. În caz contrar, există un nivel de cuplare în interiorul arhitecturii care trebuie rezolvat. Abilitatea de a livra în mod constant este un avantaj al arhitecturii microserviciilor; orice constrângere ar trebui înlăturată, la fel de mult cum dezvoltatorii rezolvă erori în aplicațiile lor.

- **Creat pentru gestiunea posibilelor erori (Build for failure)** - **Dacă ceva poate merge prost, va merge prost (Murphy)** - Nu contează câte teste sunt realizate, câte alerte pot fi declanșate; dacă microserviciul „pică", dezvoltatorii trebuie să ia în calcul acea posibilă eroare, să o trateze pe cât de elegant posibil și să definească cum se poate corecta (recovery). Când se proiectează un microserviciu, se au în vedere următoarele arii:

    - *Upstream* = înțelegerea felului în care dezvoltatorii o să trimită sau nu notificări de eroare clienților, ținând totodată cont de evitarea cuplării

    - *Downstream* = cum vor gestiona dezvoltatorii defectarea unui microserviciu sau a unui sistem (precum o bază de date) de care depind

    - *Logging* = afișarea tuturor erorilor într-un fișier de log, ținând cont de cât de des se realizează salvarea acestor informații, de cantitatea de date și cum pot fi acestea accesate. De asemenea, trebuie luate în considerare și cazuri speciale, cum ar fi informații sensibile și implicații de performanță.

    - *Monitoring* = Monitorizarea trebuie să fie proiectată cu mare atenție. Este foarte dificil de gestionat o eroare fără informațiile potrivite în sistemele de monitorizare. Dezvoltatorii trebuie să determine ce elemente ale aplicației au informații semnificative.

    - *Alerting* = presupune înțelegerea căror semnale pot indica faptul că ceva nu este în regulă, legătura semnalelor cu sistemul de monitorizare și logging-ul.

    - *Recovery* = proiectarea modului în care se revine (în urma unor erori) într-o stare normală. Revenirea automată (automatic recovery) este ideală, dar având în vedere că aceasta poate eșua, nu trebuie evitată revenirea manuală (manual recovery).

    - *Fallbacks* = Un mecanism bun de tratare a erorilor permite ca aplicația să funcționeze în continuare după apariția unei erori în sistem, în timp de dezvoltatorii lucrează să rezolve problema respectivă.

- **Scalabilitate (Scalability)** - Microserviciile trebuie să fie scalabile independent. Dacă este nevoie să se mărească numărul de cereri care poate fi gestionar, sau câte înregistrări poti fi stocate, acestea trebuie făcute în izolare. Se evită scalarea aplicației prin scalarea mai multor componente, impusă de o cuplare mare.

- **Automatizarea (Automation)** - Microserviciile trebuie proiectate ținând cont de lanțul specific CI/CD, de la construire și testare până la instalare și monitorizare. Modelul pentru dezvoltare/integrare continuă CI/CD trebuie proiectat de la începutul arhitecturii.

Domain-Driven Design (DDD)

Proiectarea bazată pe analiza domeniului reprezintă o manieră pentru dezvoltarea aplicațiilor complexe prin conectarea continuă a implementării la un model (care evoluează continuu) a conceptelor business de bază.

Premisele DDD:

• accentul principal al proiectului cade pe domeniul de bază (core domain) și pe logica domeniului (domain logic)

• Proiectele mai complexe trebuie bazate pe un model

• inițierea unei colaborări creative între experții tehnici și experții din domeniu pentru a ajunge din ce în ce mai aproape de modelul conceptual al problemei

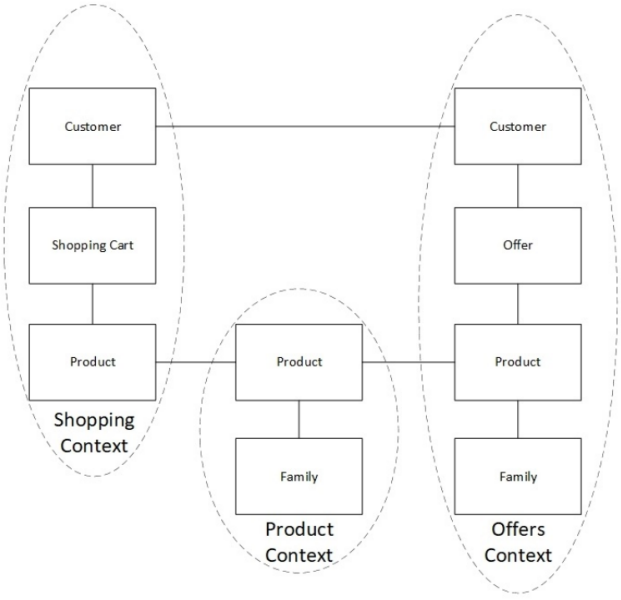
Problema: când complexitatea scapă de sub control, software-ul nu mai poate fi înțeles suficient de bine pentru a putea fi schimbat sau extins cu ușurință. Dacă complexitatea domeniului nu este tratată în proiectare, nu contează că tehnologia infrastructurii suport este bine concepută.

Principiile de proiectare:

• **Context mărginit (Bounded context):** Când se abordează un sistem complex, de obicei se abstractizează într-un model care descrie aspectele diferite ale sistemului și cum poate fi folosit pentru a rezolva probleme. Când există mai multe modele, iar codul de bază al diferitelor modele este combinat, software-ul devine plin de erori (buggy), nesigur și greu de înțeles. În DDD, se definește contextul în care se aplică un model, se stabilesc explicit granițele în ceea ce privește organizarea echipei și utilizarea în anumite părți ale aplicației, păstrând modelul **consecvent** cu aceste limite.

• **Limbaj generic specific (Ubiquitous language):** În DDD trebuie alcătuit un limbaj comun și riguros între dezvoltatori și utilizatori. Acest limbaj trebuie să fie bazat pe modelul de domeniu, ajutând în a avea o conversație generală între toți experții din domeniu, acest lucru fiind esențial la abordarea testării.

• **Capturarea/maparea contextului (Context mapping):** Într-o aplicație de dimensiuni mari, proiectată pentru mai multe contexte mărginite (bounded contexts), se poate pierde vederea de ansamblu. Inevitabil, contextele mărginite vor fi nevoite să comunice date între ele. O mapare de context este o vedere de ansamblu (global view) asupra sistemului ca un întreg, care ilustrează maniera în care contextele mărginite ar trebui să comunice între ele.



Exemplu de mapare de context

Pentru mai multe detalii, vezi cartea **„Domain-Driven Design”** scrisă de **Eric Evans**, precum și comunitatea DDD: https://dddcommunity.org/.

Folosirea DDD în microservicii

• **Bounded Context** - Nu trebuie creat un microserviciu care include mai mult de un context mărginit

• **Ubiquitous Language** - Dezvoltatorii trebuie să se asigure că maniera de comunicare utilizată este suficient de general valabil, astfel încât operațiile și interfețele care sunt expuse să fie exprimate utilizând limbajul domeniului context

• **Context Model** - Modelul utilizat de microserviciu trebuie definit într-un context mărginit și să folosească un limbaj generic (ubiquitous language), chiar și pentru entități care nu sunt expuse în nici o interfață pe care o oferă microserviciul

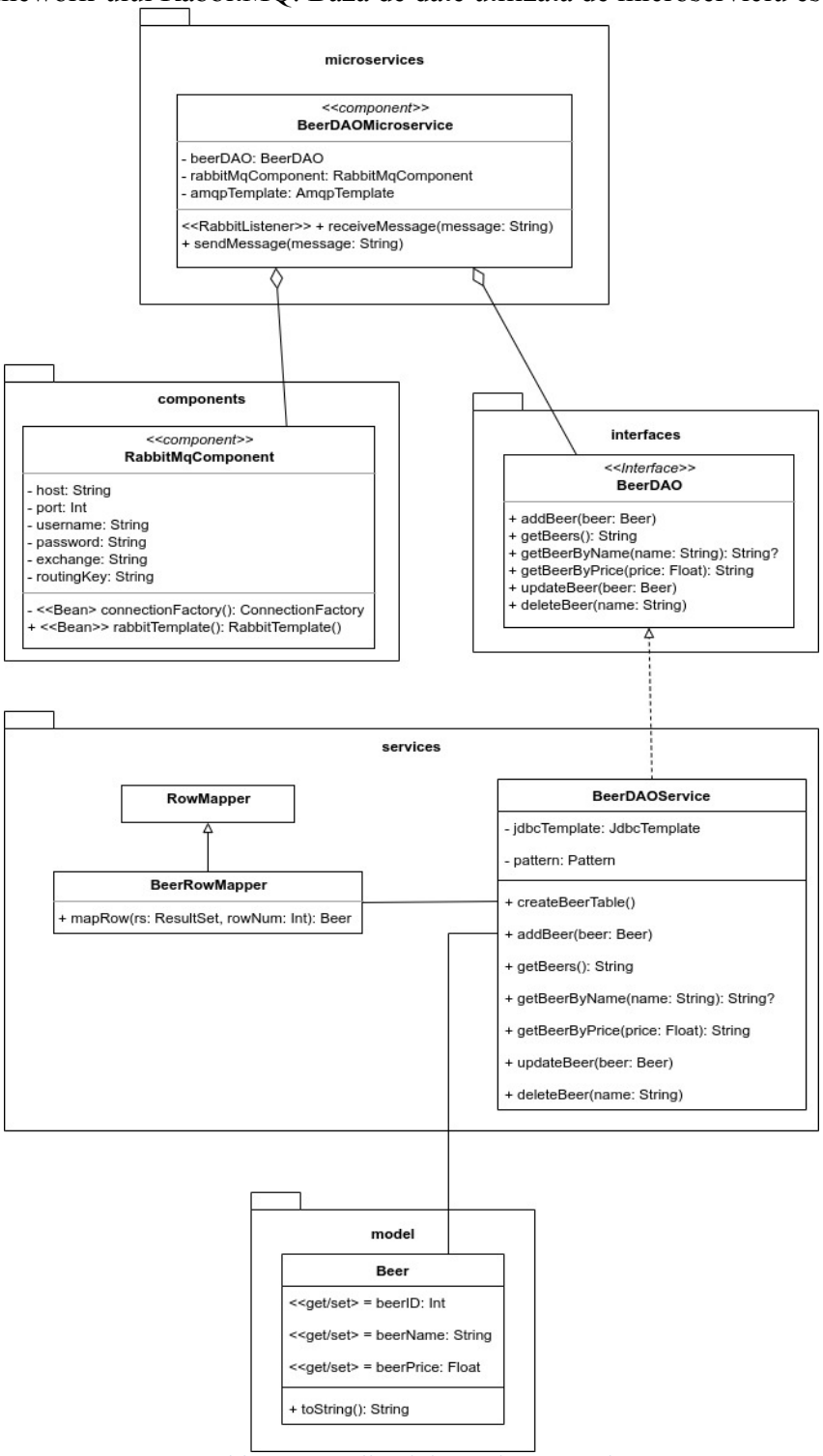
• **Context Mapping** - Trebuie examinat contextul mărginit al întregului sistem pentru a înțelege dependențele și cuplarea microserviciilor.

Exemple

Pentru un exemplu de microservicii în Kotlin cu framework-ul **Ktor**, se recomandă parcurgerea modelului de la adresa: https://dzone.com/articles/kotlin-microservices-with-ktor

Exemplul 1: SQLiteExample

Cerință: Să se creeze un microserviciu în Kotlin care să gestioneze o bază de date cu tipurile de bere. Aplicația Kotlin va comunica cu o interfață de tip CLI din python prin intermediul framework-ului RabbitMQ. Baza de date utilizată de microserviciu este SQLite3.



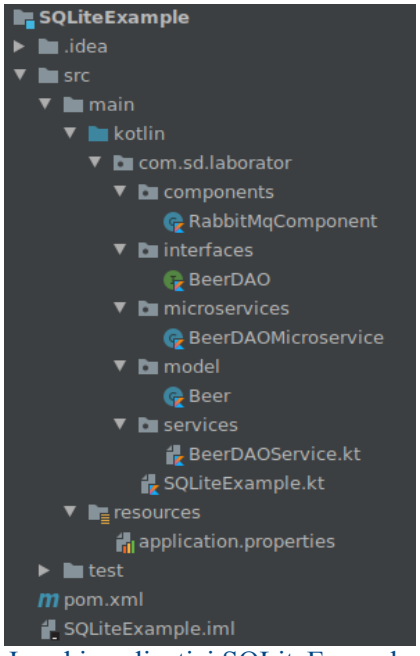
Arhitectura aplicației SQLiteExample

Exemplul 1: Configurări

Similar cu laboratorul 5 de la disciplina Sisteme Distribuite, se configurează următoarele cozi de mesaje, exchange-uri și chei de rutare în interfața RabbitMQ (localhost:15672):

- un exchange: **sqliteexample.direct**
- două cozi
  - **sqliteexample.queue**
  - **sqliteexample.queue1**
- două binding-uri:
  - sqliteexample.queue -> sqliteexample.direct, **sqliteexample.routingkey**
  - sqliteexample.queue1 -> sqliteexample.direct, **sqliteexample.routingkey1**

Structura proiectului



Ierarhia aplicației SQLiteExample

Configurarea parametrilor din fișierul appplication.properties

```
spring.datasource.url=jdbc:sqlite:beer.db
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=student
spring.rabbitmq.password=student
sqliteexample.rabbitmq.queue=sqliteexample.queue1
sqliteexample.rabbitmq.exchange=sqliteexample.direct
sqliteexample.rabbitmq.routingkey=sqliteexample.routingkey
```

Se remarcă parametrul **spring.datasource.url** care precizează faptul că jdbc-ul (java database connector) va utiliza o bază de date de tip sqlite denumită beer.db.

Crearea proiectului

Proiectul se creează conform instrucțiunilor din laboratorul 5 de la disciplina Sisteme Distribuite.

Se adaugă dependențele pentru **sqlite** în fișierul pom.xml**:**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
```

```
</dependency>
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.28.0</version>
</dependency>
```

Exemplul 1: Codul sursă

- **SQLiteExample.kt**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class SQLiteExample

fun main(args: Array<String>) {
    runApplication<SQLiteExample>(*args)
}
```

- **Beer.kt**

```
package com.sd.laborator.model

class Beer(private var id: Int, private var name: String, private var
price: Float) {

    var beerID: Int
        get() {
            return id
        }
        set(value) {
            id = value
        }
    var beerName: String
        get() {
            return name
        }
        set(value) {
            name = value
        }
    var beerPrice: Float
        get() {
            return price
        }
        set(value) {
            price = value
        }

    override fun toString(): String {
        return "Beer [id=$beerID, name=$beerName, price=$beerPrice]"
```

```
    }
}
```

- **BeerDAO.kt**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Beer

interface BeerDAO {
    // Create
    fun createBeerTable()
    fun addBeer(beer: Beer)

    // Retrieve
    fun getBeers(): String
    fun getBeerByName(name: String): String?
    fun getBeerByPrice(price: Float): String?

    // Update
    fun updateBeer(beer: Beer)

    // Delete
    fun deleteBeer(name: String)
}
```

- **BeerDAOService.kt**

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.BeerDAO
import com.sd.laborator.model.Beer
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.RowMapper
import org.springframework.stereotype.Service
import java.sql.ResultSet
import java.sql.SQLException
import java.util.regex.Pattern

class BeerRowMapper : RowMapper<Beer?> {
    @Throws(SQLException::class)
    override fun mapRow(rs: ResultSet, rowNum: Int): Beer {
        return Beer(rs.getInt("id"), rs.getString("name"),
rs.getFloat("price"))
    }
}

@Service
class BeerDAOService: BeerDAO {
    //Spring Boot will automatically wire this object using
application.properties:
    @Autowired
    private lateinit var jdbcTemplate: JdbcTemplate
    var pattern: Pattern = Pattern.compile("\\W")
```

```
    override fun createBeerTable() {
        jdbcTemplate.execute("""CREATE TABLE IF NOT EXISTS beers(
                                        id INTEGER PRIMARY KEY
AUTOINCREMENT,
                                        name VARCHAR(100) UNIQUE,
                                        price FLOAT)""")
    }

    override fun addBeer(beer: Beer) {
        if(pattern.matcher(beer.beerName).find()) {
            println("SQL Injection for beer name")
            return
        }
        jdbcTemplate.update("INSERT INTO beers(name, price) VALUES (?,
?)", beer.beerName, beer.beerPrice)
    }

    override fun getBeers(): String {
        val result: MutableList<Beer?> = jdbcTemplate.query("SELECT *
FROM beers", BeerRowMapper())
        var stringResult: String = ""
        for (item in result) {
            stringResult += item
        }
        return stringResult
    }

    override fun getBeerByName(name: String): String? {
        if(pattern.matcher(name).find()) {
            println("SQL Injection for beer name")
            return null
        }
        val result: Beer? = jdbcTemplate.queryForObject("SELECT * FROM
beers WHERE name = '$name'", BeerRowMapper())
        return result.toString()
    }

    override fun getBeerByPrice(price: Float): String {
        val result: MutableList<Beer?> = jdbcTemplate.query("SELECT *
FROM beers WHERE price <= $price", BeerRowMapper())
        var stringResult: String = ""
        for (item in result) {
            stringResult += item
        }
        return stringResult
    }

    override fun updateBeer(beer: Beer) {
        if(pattern.matcher(beer.beerName).find()) {
            println("SQL Injection for beer name")
            return
        }
```

```
        jdbcTemplate.update("UPDATE beers SET name = ?, price = ?
WHERE id = ?", beer.beerName, beer.beerPrice, beer.beerID)
    }

    override fun deleteBeer(name: String) {
        if(pattern.matcher(name).find()) {
            println("SQL Injection for beer name")
            return
        }
        jdbcTemplate.update("DELETE FROM beers WHERE name = ?", name)
    }
}
```

- **BeerDAOMicroservice.kt**

```
package com.sd.laborator.microservices

import com.sd.laborator.components.RabbitMqComponent
import com.sd.laborator.interfaces.BeerDAO
import com.sd.laborator.model.Beer
import org.springframework.amqp.core.AmqpTemplate
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Component

@Component
class BeerDAOMicroservice {
    @Autowired
    private lateinit var beerDAO: BeerDAO

    @Autowired
    private lateinit var rabbitMqComponent: RabbitMqComponent

    private lateinit var amqpTemplate: AmqpTemplate

    @Autowired
    fun initTemplate() {
        this.amqpTemplate = rabbitMqComponent.rabbitTemplate()
    }
    //citesc din queue1
    // scriu in queue
    @RabbitListener(queues = ["\${sqliteexample.rabbitmq.queue}"])
    fun recieveMessage(msg: String) {
        val processed_msg = (msg.split(",").map { it.toInt().toChar()
}).joinToString(separator="")
        val (operation, parameters) = processed_msg.split('~')
        var beer: Beer? = null
        var price: Float? = null
        var name: String? = null

        // id=1;name=Corona;price=3.6
        if("id=" in parameters) {
            println(parameters)
            val params: List<String> = parameters.split(';')
```

```
            try {
                beer = Beer(
                    params[0].split('=')[1].toInt(),
                    params[1].split('=')[1],
                    params[2].split('=')[1].toFloat()
                )
            } catch (e: Exception) {
                print("Error parsing the parameters: ")
                println(params)
                return
            }
        } else if ("price=" in parameters) {
            price = parameters.split('=')[1].toFloat()
        } else if ("name=" in parameters) {
            name = parameters.split("=")[1]
        }
        println(parameters)
        println(name)
        println(price)
        println(beer)
        var result: Any? = when(operation) {
            "createBeerTable" -> beerDAO.createBeerTable()
            "addBeer" -> beerDAO.addBeer(beer!!)
            "getBeers" -> beerDAO.getBeers()
            "getBeerByName" -> beerDAO.getBeerByName(name!!)
            "getBeerByPrice" -> beerDAO.getBeerByPrice(price!!)
            "updateBeer" -> beerDAO.updateBeer(beer!!)
            "deleteBeer" -> beerDAO.deleteBeer(name!!)
            else -> null
        }
        println("result: ")
        println(result)
        if (result != null) sendMessage(result.toString())
    }

    fun sendMessage(msg: String) {
        println("message: ")
        println(msg)

this.amqpTemplate.convertAndSend(rabbitMqComponent.getExchange(),
rabbitMqComponent.getRoutingKey(), msg)
    }
}
```

- **RabbitMqComponent**

```
package com.sd.laborator.components

import
org.springframework.amqp.rabbit.connection.CachingConnectionFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
```

```
import org.springframework.stereotype.Component

@Component
class RabbitMqComponent {
    @Value("\${spring.rabbitmq.host}")
    private lateinit var host: String
    @Value("\${spring.rabbitmq.port}")
    private val port: Int = 0
    @Value("\${spring.rabbitmq.username}")
    private lateinit var username: String
    @Value("\${spring.rabbitmq.password}")
    private lateinit var password: String
    @Value("\${sqliteexample.rabbitmq.exchange}")
    private lateinit var exchange: String
    @Value("\${sqliteexample.rabbitmq.routingkey}")
    private lateinit var routingKey: String

    fun getExchange(): String = this.exchange

    fun getRoutingKey(): String = this.routingKey

    @Bean
    private fun connectionFactory(): ConnectionFactory {
        val connectionFactory = CachingConnectionFactory()
        connectionFactory.host = this.host
        connectionFactory.username = this.username
        connectionFactory.setPassword(this.password)
        connectionFactory.port = this.port
        return connectionFactory
    }

    @Bean
    fun rabbitTemplate(): RabbitTemplate =
RabbitTemplate(connectionFactory())
}
```

Aplicația python

```
import pika
from retry import retry


class RabbitMq:
    config = {
        'host': '0.0.0.0',
        'port': 5678,
        'username': 'student',
        'password': 'student',
        'exchange': 'sqliteexample.direct',
        'routing_key': 'sqliteexample.routingkey1',
        'queue': 'sqliteexample.queue'
    }
```

```
    credentials = pika.PlainCredentials(config['username'],
                                        config['password'])
    parameters = (pika.ConnectionParameters(host=config['host']),
                  pika.ConnectionParameters(port=config['port']),
                  pika.ConnectionParameters(credentials=credentials))

    def on_received_message(self, blocking_channel,
                            deliver, properties, message):
        result = message.decode('utf-8')
        blocking_channel.confirm_delivery()
        try:
            print(result)
        except Exception:
            print("wrong data format")
        finally:
            blocking_channel.stop_consuming()

    @retry(pika.exceptions.AMQPConnectionError, delay=5, jitter=(1,
3))
    def receive_message(self):
        # automatically close the connection
        with pika.BlockingConnection(self.parameters) as connection:
            # automatically close the channel
            with connection.channel() as channel:
                channel.basic_consume(self.config['queue'],
                                      self.on_received_message)
                try:
                    channel.start_consuming()
                # Don't recover connections closed by server
                except pika.exceptions.ConnectionClosedByBroker:
                    print("Connection closed by broker.")
                # Don't recover on channel errors
                except pika.exceptions.AMQPChannelError:
                    print("AMQP Channel Error")
                # Don't recover from KeyboardInterrupt
                except KeyboardInterrupt:
                    print("Application closed.")

    def send_message(self, message):
        # automatically close the connection
        with pika.BlockingConnection(self.parameters) as connection:
            # automatically close the channel
            with connection.channel() as channel:
                self.clear_queue(channel)
                channel.basic_publish(
                             exchange=self.config['exchange'],
                             routing_key=self.config['routing_key'],
                             body=message)

    def clear_queue(self, channel):
        channel.queue_purge(self.config['queue'])
```

```
def print_menu():
    print('0 --> Exit program')
    print('1 --> addBeer')
    print('2 --> getBeers')
    print('3 --> getBeerByName')
    print('4 --> getBeerByPrice')
    print('5 --> updateBeer')
    print('6 --> deleteBeer')
    return input("Option=")


if __name__ == '__main__':
    rabbit_mq = RabbitMq()
    rabbit_mq.send_message("createBeerTable~")
    while True:
        option = print_menu()
        if option == '0':
            break
        elif option == '1':
            name = input("Beer name: ")
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "addBeer~id=-1;name={};price={}".format(name, price))
        elif option == '2':
            rabbit_mq.send_message("getBeers~")
            rabbit_mq.receive_message()
        elif option == '3':
            name = input("Beer name: ")
            rabbit_mq.send_message(
                "getBeerByName~name={}".format(name))
            rabbit_mq.receive_message()
        elif option == '4':
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "getBeerByPrice~price={}".format(price))
            rabbit_mq.receive_message()
        elif option == '5':
            id = int(input("Beer ID: "))
            name = input("Beer name: ")
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "updateBeer~id={};name={};price={}".format(id, name,
                                                           price))
            rabbit_mq.receive_message()
        elif option == '6':
            name = input("Beer name: ")
            rabbit_mq.send_message("deleteBeer~name={}".format(name))
        else:
            print("Invalid option")
```

Pentru testarea exemplului, din IntelliJ se execută **Maven -> Lifecycle -> clean + compile + package**. La final, se deschide directorul **target** creat și se execută în terminal comanda:

```
java -jar SQLiteExample-1.0.0.jar
```

Pentru a porni interfața CLI, se execută într-un terminal deschis în directorul cu aplicația python comenzile:

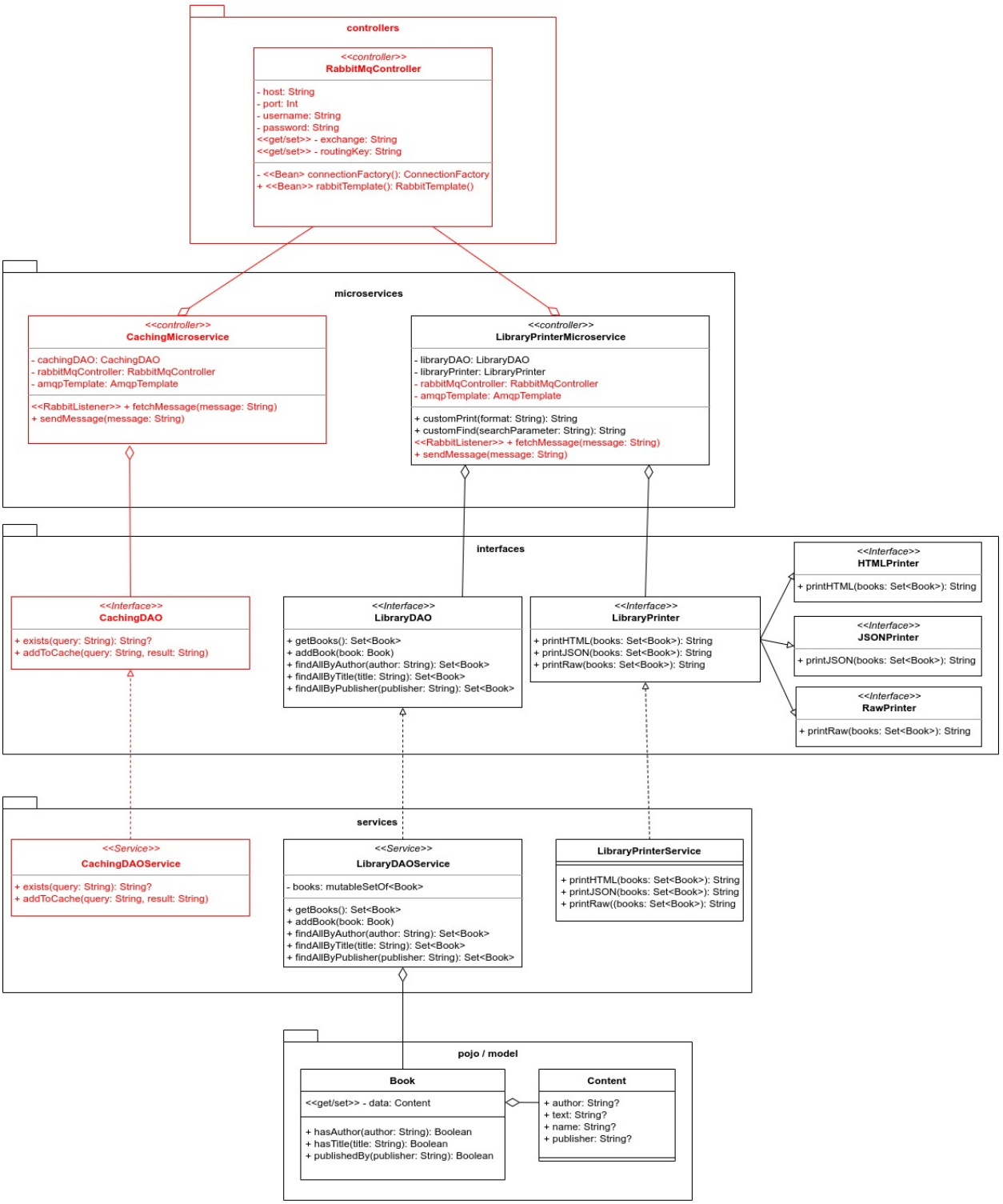
```
python3 -m venv env
source env/bin/activate
pip3 install pika==1.1.0 retry==0.9.2
python3 sqlite_example.py
```

Exemplul 2: LibraryApp

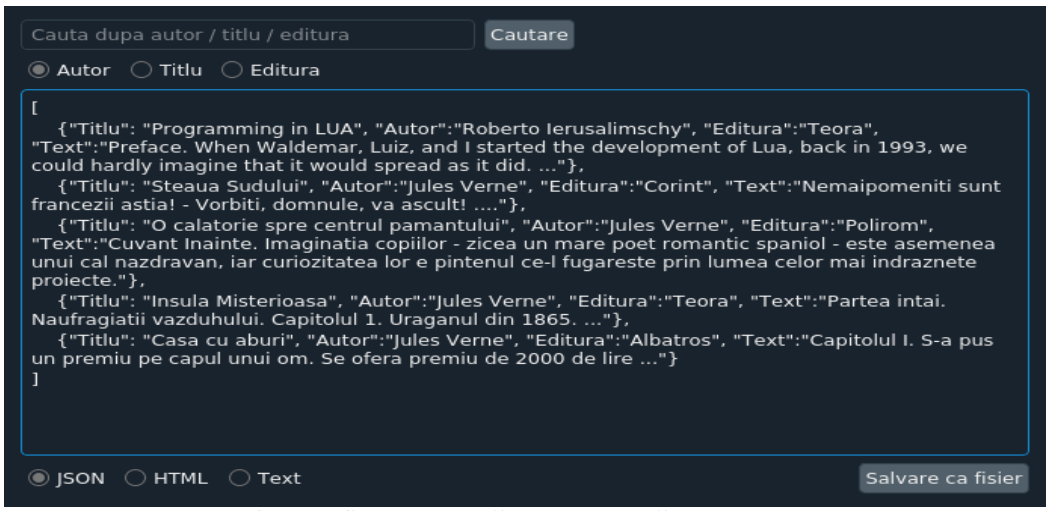
Cerință: Să se scrie un program Kotlin care să realizeze prin intermediul unui microserviciu WEB gestiunea unei biblioteci utilizând principiile SOLID. Aplicația va conține trei moduri de afișare a datelor (HTML, JSON și Raw) și va expune utilizatorului prin interfață funcționalități de tip CRUD (Create, Retrieve, Update, Delete).

Arhitectura este reprezentată în diagrama de clase de mai jos. Spre deosebire de exemplul anterior, microserviciul din LibraryApp va fi de tip Web, epunând către interfața din python metodele de afișare ale bibliotecii.

Observație: componentele cu roșu din diagrama de mai jos nu se regăsesc în exemplul curent. Acestea trebuie implementate în tema pe acasă.

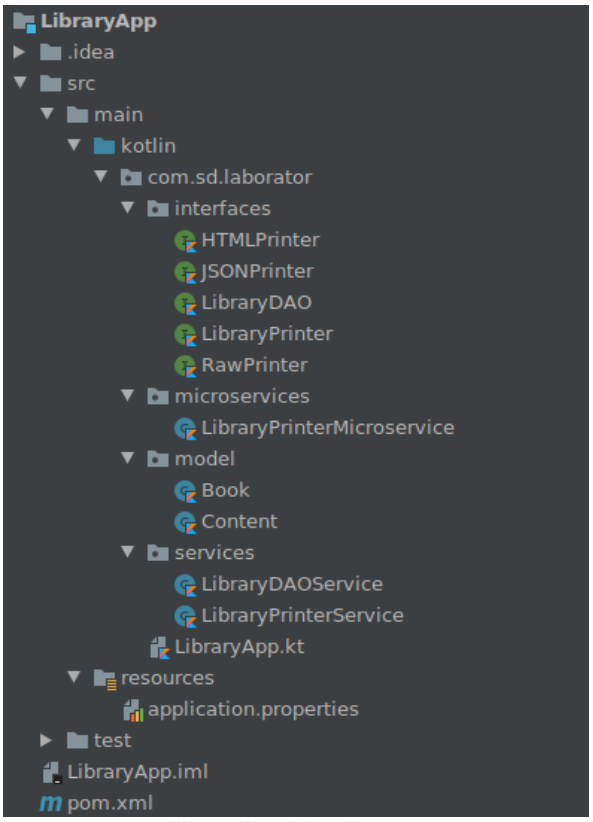


Arhitectura aplicației LibraryApp



Interfața grafică pentru LibraryApp realizată cu PyQt5

Structura proiectului



Ierarhia aplicației LibraryApp

Crearea proiectului

Proiectul se creează similar cu exemplul anterior. La final, se mai adaugă dependența de **spring-boot-starter-web** în fișierul pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Exemplul 2: Codul sursă

- **LibraryApp.kt**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication


@SpringBootApplication
class LibraryApp

fun main(args: Array<String>) {
    runApplication<LibraryApp>(*args)
}
```

- **Content.kt**

```
package com.sd.laborator.model

data class Content(var author: String?, var text: String?, var name:
String?, var publisher: String?)
```

- **Book.kt**

```
package com.sd.laborator.model

class Book(private var data: Content) {

    var name: String?
        get() {
            return data.name
        }
        set(value) {
            data.name = value
        }

    var author: String?
        get() {
            return data.author
        }
        set(value) {
            data.author = value
        }

    var publisher: String?
        get() {
            return data.publisher
        }
        set(value) {
            data.publisher = value
        }

    var content: String?
        get() {
```

```
            return data.text
        }
        set(value) {
            data.text = value
        }

    fun hasAuthor(author: String): Boolean {
        return data.author.equals(author)
    }

    fun hasTitle(title: String): Boolean {
        return data.name.equals(title)
    }

    fun publishedBy(publisher: String): Boolean {
        return data.publisher.equals(publisher)
    }
}
```

- **LibraryPrinterService.kt**

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LibraryPrinter
import com.sd.laborator.model.Book
import org.springframework.stereotype.Service

@Service
class LibraryPrinterService: LibraryPrinter {
    override fun printHTML(books: Set<Book>): String {
        var content: String = "<html><head><title>Libraria mea
HTML</title></head><body>"
        books.forEach {
            content +=
"<p><h3>${it.name}</h3><h4>${it.author}</h4><h5>${it.publisher}</h5>${
it.content}</p><br/>"
        }
        content += "</body></html>"
        return content
    }

    override fun printJSON(books: Set<Book>): String {
        var content: String = "[\n"
        books.forEach {
            if (it != books.last())
                content += "    {\"Titlu\": \"${it.name}\",
\"Autor\":\"${it.author}\", \"Editura\":\"${it.publisher}\",
\"Text\":\"${it.content}\"},\n"
            else
                content += "    {\"Titlu\": \"${it.name}\",
\"Autor\":\"${it.author}\", \"Editura\":\"${it.publisher}\",
\"Text\":\"${it.content}\"}\n"
        }
        content += "]\n"
```

```
        return content
    }

    override fun printRaw(books: Set<Book>): String {
        var content: String = ""
        books.forEach {
            content +=
"${it.name}\n${it.author}\n${it.publisher}\n${it.content}\n\n"
        }
        return content
    }
}
```

- **LibraryDAOService.kt**

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.model.Book
import com.sd.laborator.model.Content
import org.springframework.stereotype.Service

@Service
class LibraryDAOService: LibraryDAO {
    private var books: MutableSet<Book> = mutableSetOf(
        Book(Content("Roberto Ierusalimschy","Preface. When Waldemar,
Luiz, and I started the development of Lua, back in 1993, we could
hardly imagine that it would spread as it did. ...","Programming in
LUA","Teora")),
        Book(Content("Jules Verne","Nemaipomeniti sunt francezii
astia! - Vorbiti, domnule, va ascult! ....","Steaua
Sudului","Corint")),
        Book(Content("Jules Verne","Cuvant Inainte. Imaginatia
copiilor - zicea un mare poet romantic spaniol - este asemenea unui
cal nazdravan, iar curiozitatea lor e pintenul ce-l fugareste prin
lumea celor mai indraznete proiecte.","O calatorie spre centrul
pamantului","Polirom")),
        Book(Content("Jules Verne","Partea intai. Naufragiatii
vazduhului. Capitolul 1. Uraganul din 1865. ...","Insula
Misterioasa","Teora")),
        Book(Content("Jules Verne","Capitolul I. S-a pus un premiu pe
capul unui om. Se ofera premiu de 2000 de lire ...","Casa cu
aburi","Albatros"))
    )
    override fun getBooks(): Set<Book> {
        return this.books
    }

    override fun addBook(book: Book) {
        this.books.add(book)
    }

    override fun findAllByAuthor(author: String): Set<Book> {
        return (this.books.filter { it.hasAuthor(author) }).toSet()
```

```
    }

    override fun findAllByTitle(title: String): Set<Book> {
        return (this.books.filter { it.hasTitle(title) }).toSet()
    }

    override fun findAllByPublisher(publisher: String): Set<Book> {
        return (this.books.filter { it.publishedBy(publisher)
}).toSet()
    }
}
```

- **HTMLPrinter**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface HTMLPrinter {
    fun printHTML(books: Set<Book>): String
}
```

- **JSONPrinter**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface JSONPrinter {
    fun printJSON(books: Set<Book>): String
}
```

- **LibraryDAO**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface LibraryDAO {
    fun getBooks(): Set<Book>
    fun addBook(book: Book)
    fun findAllByAuthor(author: String): Set<Book>
    fun findAllByTitle(title: String): Set<Book>
    fun findAllByPublisher(publisher: String): Set<Book>
}
```

- **LibraryPrinter**

```
package com.sd.laborator.interfaces

interface LibraryPrinter: HTMLPrinter, JSONPrinter, RawPrinter
```

- **RawPrinter**

```
package com.sd.laborator.interfaces

```

```
import com.sd.laborator.model.Book

interface RawPrinter {
    fun printRaw(books: Set<Book>): String
}
```

- **LibraryPrinterMicroservice**

Se observă faptul că microserviciul are acces prin intermediul unui LibraryDAO la baza de date (pe moment, la datele hardcodate) și totodată la funcționalitățile de printare ale LibraryPrinter. Funcționalitățile microserviciului WEB sunt expuse la următoarele URL-uri:

» http://localhost:8080/print?format=html
» http://localhost:8080/find?author=Jules%20Verne
» http://localhost:8080/find?title=Steaua%20Sudului
» http://localhost:8080/find?publisher=Corint

```
package com.sd.laborator.microservices

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.interfaces.LibraryPrinter
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody

@Controller
class LibraryPrinterMicroservice {
    @Autowired
    private lateinit var libraryDAO: LibraryDAO
    @Autowired
    private lateinit var libraryPrinter: LibraryPrinter

    @RequestMapping("/print", method = [RequestMethod.GET])
    @ResponseBody
    fun customPrint(@RequestParam(required = true, name = "format",
defaultValue = "") format: String): String {
        return when(format) {
            "html" -> libraryPrinter.printHTML(libraryDAO.getBooks())
            "json" -> libraryPrinter.printJSON(libraryDAO.getBooks())
            "raw" -> libraryPrinter.printRaw(libraryDAO.getBooks())
            else -> "Not implemented"
        }
    }

    @RequestMapping("/find", method = [RequestMethod.GET])
    @ResponseBody
    fun customFind(@RequestParam(required = false, name = "author",
defaultValue = "") author: String,
                   @RequestParam(required = false, name = "title",
defaultValue = "") title: String,
                   @RequestParam(required = false, name = "publisher",
defaultValue = "") publisher: String): String {
        if (author != "")
```

```
            return
this.libraryPrinter.printJSON(this.libraryDAO.findAllByAuthor(author))
        if (title != "")
            return
this.libraryPrinter.printJSON(this.libraryDAO.findAllByTitle(title))
        if (publisher != "")
            return
this.libraryPrinter.printJSON(this.libraryDAO.findAllByPublisher(pu-
blisher))
        return "Not a valid field"
    }
}
```

Aplicația python

În codul de mai jos, se remarcă că, spre deosebire de aplicația din laboratorul precedent în care comunicația era realizată prin cozi de mesaje, exemplul curent utilizează modulul *requests* trimițând cereri HTTP de tip GET către microserviciul Web din Kotlin.

```
import os
import sys
import requests
import qdarkstyle
from requests.exceptions import HTTPError
from PyQt5.QtWidgets import (QWidget, QApplication, QFileDialog,
                             QMessageBox)
from PyQt5 import QtCore
from PyQt5.uic import loadUi

def debug_trace(ui=None):
    from pdb import set_trace
    QtCore.pyqtRemoveInputHook()
    set_trace()
    # QtCore.pyqtRestoreInputHook()

class LibraryApp(QWidget):
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))

    def __init__(self):
        super(LibraryApp, self).__init__()
        ui_path = os.path.join(self.ROOT_DIR, 'library_manager.ui')
        loadUi(ui_path, self)
        self.search_btn.clicked.connect(self.search)
        self.save_as_file_btn.clicked.connect(self.save_as_file)

    def search(self):
        search_string = self.search_bar.text()
        url = None
        if not search_string:
            if self.json_rb.isChecked():
                url = '/print?format=json'
            elif self.html_rb.isChecked():
                url = '/print?format=html'
```

```
            else:
                url = '/print?format=raw'
        else:
            if self.author_rb.isChecked():
                url = '/find?author={}'.format(
                            search_string.replace(' ', '%20'))
            elif self.title_rb.isChecked():
                url = '/find?title={}'.format(
                            search_string.replace(' ', '%20'))
            else:
                url = '/find?publisher={}'.format(
                            search_string.replace(' ', '%20'))
        full_url = "http://localhost:8080" + url
        try:
            response = requests.get(full_url)
            self.result.setText(response.content.decode('utf-8'))
        except HTTPError as http_err:
            print('HTTP error occurred: {}'.format(http_err))
        except Exception as err:
            print('Other error occurred: {}'.format(err))

    def save_as_file(self):
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        file_path = str(
            QFileDialog.getSaveFileName(self,
                                        'Salvare fisier',
                                        options=options))
        if file_path:
            file_path = file_path.split("'")[1]
            if not file_path.endswith('.json') and not
file_path.endswith(
                    '.html') and not file_path.endswith('.txt'):
                if self.json_rb.isChecked():
                    file_path += '.json'
                elif self.html_rb.isChecked():
                    file_path += '.html'
                else:
                    file_path += '.txt'
            try:
                with open(file_path, 'w') as fp:
                    if file_path.endswith(".html"):
                        fp.write(self.result.toHtml())
                    else:
                        fp.write(self.result.toPlainText())
            except Exception as e:
                print(e)
                QMessageBox.warning(self, 'Library Manager',
                                    'Nu s-a putut salva fisierul')


if __name__ == '__main__':
```

```
    app = QApplication(sys.argv)

    stylesheet = qdarkstyle.load_stylesheet_pyqt5()
    app.setStyleSheet(stylesheet)

    window = LibraryApp()
    window.show()
    sys.exit(app.exec_())
```

Pentru testarea exemplului, din IntelliJ se execută **Maven -> Lifecycle -> clean + compile**, apoi **Maven -> Plugins -> spring-boot -> spring-boot:run.**

Pentru a porni interfața grafică, se execută într-un terminal deschis în directorul cu aplicația python comenzile:

```
python3 -m venv env
source env/bin/activate
pip3 install -r requirements.txt
python3 library_manager.py
```

Aplicaţii şi teme

Aplicații de laborator:

- Să se reimplementeze stocarea datelor în exemplul LibraryApp utilizând o bază de date SQLite (ca în exemplul 1), având tabela **Book** în diagrama E-R de mai jos.

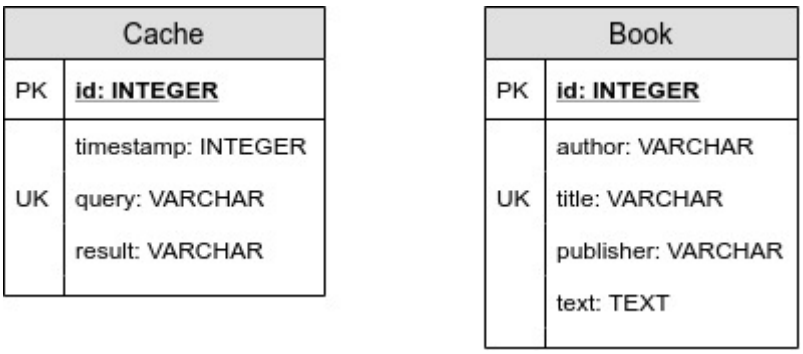


Diagrama Entitate-Relație

- Să se combine funcționalitatea de căutare cu cea de afișare într-un anumit format (HTML, JSON, raw) sub forma unui nou end-point accesibil printr-un request HTTP de tip GET la un URL: http://localhost:8080/find-and-print?author=<author-name>&format=json

Teme pe acasă:

- Să se implementeze un mecanism de caching care stochează în baza de date interogarea utilizatorului (dacă nu există deja în baza de date), rezultatul căutării pe baza interogării și timestamp-ul căutării. Vezi diagrama UML de clase, precum și diagrama E-R de mai sus pentru mai multe detalii despre implementare. Microserviciul **CachingMicroservice** trebuie să comunice prin intermediul a două cozi de mesaje cu **LibraryPrinterMicroservice**. Astfel, la fiecare interogare introdusă de utilizator, se verifică întâi cache-ul. În cazul unui HIT (interogarea a mai fost introdusă anterior), dacă timestamp-ul nu este mai vechi de o oră, se ia rezultatul din cache. Dacă timestamp-ul depășește intervalul de o oră sau în cazul unui MISS, se realizează căutarea propriu-zisă, iar rezultatul este actualizat/scris în cache.

Observație: Pentru simplitate, CachingMicroservice a fost introdus în cadrul aceluiași proiect. Totuși, pentru a respecta principiile de proiectare ale microserviciilor, trebuie creat un proiect nou pentru acesta, la final rezultând un fișier jar pentru fiecare microserviciu.