A feladatok általános követelményei

Architektúra

A feladatok tipikusan olyan egyszerű alkalmazásintegrációk, amelyek a Docker-környezetben megvalósíthatók.

Feladatok beadása

A feladatot a félév végén kell leadni, személyesen bemutatva. Lehet saját laptopon is vagy a labor gépein. Csak e-mailben elküldött megoldásokat nem fogadunk el.

Beadási határidő

Az utolsó tanítási héttel bezárólag minden gyakorlaton. Levelező képzésben a megbeszélt időpontban. Természetesen ezután pótlás is lehetséges.

Feladatok

1. Színüzenetek feldolgozása RabbitMQ-val

Készítsen Docker-konténerekben futó alkalmazást, amely RabbitMQ-üzenetsoron keresztül dolgoz fel színüzeneteket. A megoldás legalább az alábbi komponensekből álljon:

- RabbitMQ broker konténer.
- Color Producer komponens, amely 1 másodpercenként véletlenszerűen RED, GREEN vagy BLUE értékű üzenetet küld a *colorQueue* üzenetsorba.
- Három külön Consumer komponens, amelyek a RED, GREEN és BLUE üzenetek feldolgozásáért felelősek.
- Statistics Reporter komponens, amely a feldolgozási statisztikákat olvassa.

A szín szerinti szétválasztást RabbitMQ-val megvalósított routinggal vagy több queue használatával kell megoldani. A producer minden üzenetben adja meg a színértékét, a consumer-komponensek pedig csak a saját színükhöz tartozó üzeneteket dolgozzák fel. Minden consumer tartson számlálót, és minden 10 sikeresen feldolgozott azonos színű üzenet után küldjön statisztikai üzenetet a *colorStatistics* üzenetsorba.

A Statistics Reporter komponens a *colorStatistics* üzenetsorból olvassa az üzeneteket, és írja ki a konzolra a feldolgozás eredményét, például:

- *10 RED messages have been processed*
- *10 GREEN messages have been processed*
- *10 BLUE messages have been processed*

A teljes rendszert *docker compose* segítségével lehessen indítani, és a komponensek konfigurációját környezeti változókkal lehessen megadni.

---

2. Hibakezelés és dead-letter queue RabbitMQ-val

Készítsen Docker-konténerekben futó alkalmazást, amely az 1. feladat működését egészíti ki hibakezeléssel és dead-letter queue használatával.

A rendszer legalább az alábbi komponensekből álljon:

- RabbitMQ broker konténer.
- Color Producer komponens, amely 1 másodpercenként véletlenszerűen RED, GREEN vagy BLUE értékű üzenetet küld a *colorQueue* üzenetsorba.
- Három külön Consumer komponens, amelyek a RED, GREEN és BLUE üzenetek feldolgozásáért felelősek.
- Statistics Reporter komponens, amely a sikeres feldolgozásokról szóló statisztikai üzeneteket olvassa.
- Dead Letter Reporter komponens, amely a sikertelenül feldolgozott üzeneteket figyeli.

A szín szerinti szétválasztást RabbitMQ routinggal vagy több queue-val kell megoldani. A consumer komponensek csak a saját színükhöz tartozó üzeneteket dolgozzák fel. A feldolgozás során minden consumer véletlenszerűen, átlagosan 10 üzenetből 3 esetben hibát szimuláljon. Ilyenkor az üzenetet ne igazolja vissza sikeresként, hanem a RabbitMQ dead-letter mechanizmusán keresztül kerüljön a *DLQ* üzenetsorba.

Minden consumer tartson számlálót, és minden 10 sikeresen feldolgozott, azonos színű üzenet után küldjön statisztikai üzenetet a *colorStatistics* üzenetsorba. A Statistics Reporter komponens a *colorStatistics* üzenetsorból olvassa az üzeneteket, és írja ki a konzolra például:

- *10 RED messages have been processed*

A Dead Letter Reporter komponens a *DLQ* üzenetsorból olvassa a sikertelen üzeneteket, és írja ki a konzolra, hogy melyik színű üzenet feldolgozása hiúsult meg.

A teljes rendszert *docker compose* segítségével lehessen indítani. A dead-letter queue konfigurációját a RabbitMQ queue- és exchange-beállításaiban kell megadni, nem alkalmazásszintű külön hibasorba küldéssel.

---

3. SOAP szolgáltatás RabbitMQ producerként

Készítsen Docker-konténerekben futó alkalmazást, amely az 1. feladat működését egészíti ki SOAP-alapú bemeneti szolgáltatással. A kliens ebben a feladatban nem kapcsolódhat közvetlenül a RabbitMQ brokerhez, hanem SOAP-hívásokon keresztül küldi a színüzeneteket egy külön szolgáltatásnak.

A rendszer legalább az alábbi komponensekből álljon:

- RabbitMQ broker konténer.
- SOAP Color Gateway szolgáltatás, amely SOAP végponton fogad RED, GREEN vagy BLUE értékű színüzeneteket, majd továbbítja őket a *colorQueue* üzenetsorba.
- SOAP Client komponens, amely 1 másodpercenként véletlenszerű színt küld a SOAP Color Gateway szolgáltatásnak.
- Három külön Consumer komponens, amelyek a RED, GREEN és BLUE üzenetek feldolgozásáért felelősek.
- Statistics Reporter komponens, amely a feldolgozási statisztikákat olvassa.

A szín szerinti szétválasztást RabbitMQ routinggal vagy több queue-val kell megoldani. A consumer komponensek csak a saját színükhöz tartozó üzeneteket dolgozzák fel. Minden consumer tartson számlálót, és minden 10 sikeresen feldolgozott, azonos színű üzenet után küldjön statisztikai üzenetet a *colorStatistics* üzenetsorba.

A Statistics Reporter komponens a *colorStatistics* üzenetsorból olvassa az üzeneteket, és írja ki a konzolra például:

- *10 RED messages have been processed*

A teljes rendszert *docker compose* segítségével lehessen indítani. A bemutatás során legyen egyértelműen látható, hogy a külső kliens csak a SOAP-szolgáltatással kommunikál, a RabbitMQ-kapcsolatot kizárólag a szerveroldali komponensek használják.

---

4. SOAP bemenet hibakezeléssel és dead-letter queue-val

Készítsen Docker-konténerekben futó alkalmazást, amely a 2. feladat működését egészíti ki SOAP-alapú bemeneti szolgáltatással. A kliens ebben a feladatban sem kapcsolódhat közvetlenül a RabbitMQ brokerhez, hanem SOAP-hívásokon keresztül küldi a színüzeneteket egy külön szolgáltatásnak.

A rendszer legalább az alábbi komponensekből álljon:

- RabbitMQ broker konténer.
- SOAP Color Gateway szolgáltatás, amely SOAP végponton fogad RED, GREEN vagy BLUE értékű színüzeneteket, majd továbbítja őket a *colorQueue* üzenetsorba.
- SOAP Client komponens, amely 1 másodpercenként véletlenszerű színt küld a SOAP Color Gateway szolgáltatásnak.
- Három külön Consumer komponens, amelyek a RED, GREEN és BLUE üzenetek feldolgozásáért felelősek.

- Statistics Reporter komponens, amely a sikeres feldolgozásokról szóló statisztikai üzeneteket olvassa.
- Dead Letter Reporter komponens, amely a sikertelenül feldolgozott üzeneteket figyeli.

A szín szerinti szétválasztást RabbitMQ routinggal vagy több queue-val kell megoldani. A consumer komponensek csak a saját színükhöz tartozó üzeneteket dolgozzák fel. A feldolgozás során minden consumer véletlenszerűen, átlagosan 10 üzenetből 3 esetben hibát szimuláljon. Ilyenkor az üzenetet ne igazolja vissza sikeresként, hanem a RabbitMQ dead-letter mechanizmusán keresztül kerüljön a *DLQ* üzenetsorba.

Minden consumer tartson számlálót, és minden 10 sikeresen feldolgozott, azonos színű üzenet után küldjön statisztikai üzenetet a *colorStatistics* üzenetsorba. A Statistics Reporter komponens a *colorStatistics* üzenetsorból olvassa az üzeneteket, a Dead Letter Reporter komponens pedig a *DLQ* üzenetsorból jelzi a sikertelen feldolgozásokat.

A teljes rendszert *docker compose* segítségével lehessen indítani. A bemutatás során legyen látható a SOAP-hívás, a sikeres üzenetfeldolgozás és a dead-letter queue-ba kerülő hibás üzenet is.

---

5. Párhuzamos feldolgozás competing consumer mintával

Készítsen Docker konténerekben futó alkalmazást, amely RabbitMQ work queue és több párhuzamos consumer segítségével dolgoz fel színüzeneteket.

A rendszer legalább az alábbi komponensekből álljon:

- RabbitMQ broker konténer.
- Color Producer komponens, amely 0.5 másodpercenként véletlenszerű RED, GREEN vagy BLUE szöveges üzenetet küld a *colorQueue* üzenetsorba.
- Három Worker komponens, amelyek ugyanarról a *colorQueue* üzenetsorról olvasnak competing consumer mintával.
- Processed Color Reporter komponens, amely a feldolgozott üzenetekről szóló értesítéseket olvassa.

A három Worker komponens folyamatosan figyelje a *colorQueue* üzenetsort. Ha egy Worker üzenetet kap, szimuláljon feldolgozási időt véletlenszerű, 2000 és 9000 milliszekundum közötti várakozással, majd igazolja vissza az üzenet sikeres feldolgozását. A Worker komponensek közvetlenül ne írjanak a konzolra.

Minden sikeres feldolgozás után a Worker küldjön üzenetet a *processedColors* üzenetsorba arról, hogy melyik Worker melyik színt dolgozta fel. A Processed Color Reporter komponens a *processedColors* üzenetsorból olvassa az üzeneteket, és írja ki a konzolra például:

- *Client 2 processed a BLUE message*

A teljes rendszert *docker compose* segítségével lehessen indítani. A bemutatás során legyen látható, hogy a *colorQueue* üzeneteit a három Worker megosztva dolgozza fel, és a hosszabb feldolgozási idejű Worker nem akadályozza meg a többi Worker további munkáját.

6. Prepaid mobilfeltöltő rendszer TCP socket kapcsolattal

Készítsen egy képzeletbeli prepaid mobilfeltöltő alkalmazást, amely egy TCP alapú socket szerver lesz egy adott porton fogadja a kliensek kéréseit. A kliensek képzeletbeli ATM-automaták, amelyek a következő protokoll szerint működnek: 1.) a kliens egy tesztüzenetben elküldi a feltölteni kívánt telefonszámot, 2.) ha a szám feltölthető, akkor OK egyébként ERROR üzenettel megszakítja a tranzakciót. OK-üzenet esetén egy tranzakció azonosítót kapunk, amely a következő 3. lépésben azonosítja a folyamatot. 3.) a kliens a tényleges feltöltést kezdeményezi a korábban kapott tranzakció azonosítóval, de újra el kell küldeni a telefonszámot és a feltöltési összeget.

A lehetséges feltöltési összegek: 3000,5000,10000,15000 Készítsen egy lekérdező klienst, amely egy adott telefonszámhoz tartozó korábbi tranzakciókat listázza. Használjon adatbázist a tranzakciók tárolására a szerveralkalmazásban.

Indítson 3 klienst, amelyek egy előre adott telefonszám-halmazt használnak és véletlenszerűen teszt- és feltöltéstranszakciókat indítanak.

7. Üzenetsor vezérelt fájlfeldolgozó rendszer bővíthető formátumokkal

Készítsen egy kliensalkalmazást, ami HTTP POST-kérésekkel szöveges adatokat küld egy üzenetsorvezérelt mintarendszerbe. Úgy tervezze meg az alaprendszert, hogy később nemcsak szöveges, hanem más, pl. bináris állományokat is tudjon kezelni. A mintarendszer alapfeladata, hogy megszámolja a szöveges állományokban a szavak számát és eredményként visszaadja a kliensnek. Készítsen egy futtató környezetet, amiben több kliens párhuzamosan, véletlenszerű állományokkal használja a szolgáltatást. Készítsen legalább 3 mintaszöveget a teszteléshez. Tételezzük fel, hogy a működő rendszert ki kell egészíteni egy új funkcióval: most már bmp-formátumú képeket is lehet küldeni, amelyeknél a rendszer a kép méretét adja vissza eredményként. Készítse el úgy a rendszert, hogy menet közben lehessen kicserélni a kliens és szerverkomponenseket, azaz nem lehet leállítani a rendszert. A megoldáshoz használja az itt bemutatott elveket: mintapélda

8. SQL adatbázis elérése SOAP szolgáltatáson keresztül

Adott egy A és egy B szerverkomponens. A B komponens hozzáfér egy SQL-adatbázishoz, amiben 1 db tábla van, ami személyek adatait tartalmazza (név, születési idő, stb..). Az 'A' komponensnek nincs hozzáférése az adatbázishoz. Az 'A' komponens rendelkezik viszont egy böngészőben megjelenő felülettel, ahol személyek adatait lehet lekérdezni (összes személyt egyszerre) és új személyt lehet felvenni. A 'B' komponens két funkciója: egy új személy felvétele és a személyek lekérdezése az adatbázisból. Készítse el az A és B komponenseket és hozzon létre kapcsolatot közöttük SOAP-felhasználásával. Az A és B komponensek tetszőleges technológiák lehetnek.

9. NoSQL adatbázis elérése REST szolgáltatáson keresztül

Adott egy A és egy B szerverkomponens. A B komponens hozzáfér egy NoSQL adatbázishoz, amiben 1 db tábla van, ami könyvek adatait tartalmazza (cím, szerzők, kiadó, év, stb..). Az 'A' komponensnek nincs hozzáférése az adatbázishoz. Az 'A' komponens rendelkezik viszont egy böngészőben megjelenő felülettel, ahol könyvek adatait lehet lekérdezni (összes könyvet egyszerre) és új könyvet is fel lehet venni. A 'B' komponens két funkciója: egy új könyv felvétele és a könyvek lekérdezése az adatbázisból. Készítse el az A és B komponenseket és hozzon létre kapcsolatot közöttük JAX/RS felhasználásával. Az A és B komponensek tetszőleges technológiák lehetnek.

10. Verzióváltás működés közben üzenetsor alapú rendszerben

Készítsen olyan mintarendszert, amely képes működés közben verziót váltani. Adott egy klienskomponens (A), ami 2 másodpercenként üzenetet küld a *tasks* üzenetsornak. A kezdeti üzenetek verziója V1.0. Készítsen egy consumer komponenst (B), ami feldolgozza az üzeneteket és a *finishedTasks* üzenetsorra küldi a kész üzeneteket, az üzenetekhez hozzáfűzi az aktuális időt. A B komponens fel legyen készítve arra az esetre, ha az üzenet verziószáma nem V1.0, ekkor az üzenetet az invalidTasks üzenetsorra továbbítja. A (C) consumer komponens az *invalidTasks* üzenetsorról leveszi az üzeneteket és 5 másodperc késleltetéssel visszateszi a *tasks* üzenetsorra. Mutassuk be, ha a (A) komponens módosításával V2.0 verziószámú üzeneteket küldünk és a (B) komponensben feldogozzuk a V2.0 üzeneteket (azaz nem küldjük az *invalidTasks* sorra őket), akkor a rendszer kis késleltetéssel, de leállítás nélkül tud üzemelni.

11. Fény intenzitás monitorozó rendszer

Készítsen egy alkalmazást, amely a beltéri fényszintet (lux) figyeli. A rendszer három külön kliensből áll: egy adatgenerátorból, egy feldolgozóból és egy riasztás-kezelő kliensből.

Komponens 1: Light Intensity Generation Client

1. Csatlakozás: A kliens a *lightIntensityQueue* pontról-pontra típusú (point-to-point) üzenetsorhoz csatlakozik.
2. Feladat: 3 másodpercenként véletlenszerű fényszint-adatokat (lux-értékeket) küld, például 0 és 2000 lux között.
    1. Példa: 300 lux, 1500 lux, stb.

Komponens 2: Light Intensity Processor

1. Üzenetfogadás: Egy processzor, amely kizárólag a *lightIntensityQueue* üzeneteit kapja meg

(fényszintmérési adatok).
2. Feldolgozás: Meghatározza, hogy a fényerő értéke túl alacsony-e. Például dönthetünk úgy, hogy 100 lux alatti érték esetén „sötét” állapotot észlelünk.
3. Riasztás küldése: Ha 3 egymást követő mérés alatt marad 100 luxon, a processzor egy riasztásüzenetet küld a *lightAlertQueue* üzenetsorba azzal a szöveggel, hogy pl. “Low light alert: 3 consecutive readings below 100 lux.”

Komponens 3: Alert Reporting Client

1. Fogyasztás: A kliens a *lightAlertQueue* üzenetsorból olvassa a riasztásokat.
2. Kimenet: A kapott értesítéseket kiírja a konzolra, pl. “Low light alert: 3 consecutive readings below 100 lux.”

Működés tesztelése:

- Üzenetküldés és -fogadás tesztelése: Ellenőrizd, hogy a fényszint-adatokat helyesen küldi és fogadja a rendszer.
- Alacsony fényszint felismerése: Teszteld, hogy 3 egymást követő <100 lux érték esetén a riasztás helyesen továbbítódik.
- Riasztási mechanizmus: Ellenőrizd, hogy a lightAlertQueue-ba kerül-e az üzenet, és a kliens kiírja-e a figyelmeztetést a konzolra.

---

12. Hangerő figyelő rendszer (Sound Level Alert System)

Készítsen egy alkalmazást, amely zajszint (decibel) értékeket figyel egy adott környezetben. Három kliensből áll: egy adatokat generáló kliensből, egy zajszint-feldolgozóból és egy riasztási kliensből.

Komponens 1: Sound Level Generation Client

1. Csatlakozás: A kliens a *soundLevelQueue* pontról-pontra típusú üzenetsorhoz csatlakozik.
2. Feladat: 2 másodpercenként küld véletlenszerű zajszint-adatokat decibelben, pl. 30 dB és 120 dB között.

Komponens 2: Sound Level Processor

1. Üzenetfogadás: kizárólag a *soundLevelQueue*-ból olvas.
2. Feldolgozás: Meghatározza, hogy a zajszint magas-e. Például 80 dB felett „túl hangos” értéknek tekintjük.
3. Riasztás: Ha egy időn belül 5 db “túl hangos” érték (80 dB felett) érkezik, küld egy üzenetet a *soundAlertQueue*-ba: “High noise alert: 5 high decibel readings detected.”
    1. Megjegyzés: Dönthetünk úgy, hogy egymást követő 5 mérés is elegendő, vagy a feldolgozó számlálja, amíg össze nem gyűlik 5 hangos mérés, majd riaszt.

Komponens 3: Alert Reporting Client

1. Fogyasztás: Feliratkozik a *soundAlertQueue* üzenetsorra.
2. Kimenet: Kiírja a konzolra: “High noise alert: 5 high decibel readings detected.”

Tesztelési feladat:

- Üzenetküldés és -fogadás tesztelése: Győződjön meg róla, hogy a decibelértékek áramlása zavartalan.
- Magas zajszint azonosítása: Ellenőrizze, hogy 80 dB feletti mérésekből 5 után riasztás jön létre.
- Riasztási üzenet: Vizsgálja meg, hogy megjelenik-e a kívánt üzenet a *soundAlertQueue*-ban és a konzolon.

---

13. Nyomás figyelő rendszer (Pressure Monitoring System)

Készítsen egy alkalmazást, amely folyadék- vagy gáznyomás-adatokat figyel egy rendszerben. A feladat során három klienssel dolgozunk: egy nyomásgeneráló klienssel, egy nyomást figyelő processzorral és egy riasztást kiértékelő klienssel.

Komponens 1: Pressure Generation Client

1. Csatlakozás: A kliens a *pressureQueue* pontról-pontra típusú üzenetsorhoz csatlakozik.
2. Feladat: 4 másodpercenként küld véletlenszerű nyomásértékeket (pl. 0 és 10 bar között).

Komponens 2: Pressure Alert Processor

1. Üzenetfogadás: kizárólag a *pressureQueue* üzeneteit kapja.
2. Feldolgozás: Meghatározza, hogy a nyomás értéke veszélyesen magas-e (például 8 bar felett).
3. Riasztás küldése: Ha 2 egymást követő mérés 8 bar felett van, akkor a processzor riasztást küld a *pressureAlertQueue* üzenetsorba: “High pressure alert: 2 consecutive readings above 8 bar detected.”

Komponens 3: Alert Reporting Client

1. Fogyasztás: A kliens olvassa a *pressureAlertQueue* üzeneteit.
2. Kimenet: Konzolra írja a riasztás szövegét, pl. “High pressure alert: 2 consecutive readings above 8 bar detected.”

Tesztek

- Üzenetek küldése és fogadása: Ellenőrizze, hogy a nyomásértékek megfelelően jutnak el az első kliensből a processzorig.
- Magas nyomás azonosítása: Tesztelje, hogy a processzor 8 bar felett helyesen detektálja-e a veszélyes értékeket.
- Consecutive readings logika: Vizsgálja meg, hogy pontosan akkor jön-e létre riasztás, ha 2 egymást követő mérés magas.
