IMAP postafiókból levél letöltő alkalmazás terv

# Szükséges technológiák

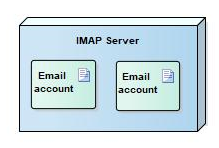
Az alkalmazás működéséhez a következő technológiák és eszközök szükségesek:

* **Imap-downloader alkalmazás**:
  + Java 8 (vagy újabb)
  + Spring Framework
  + Spring Data JPA (adatbázis kapcsolat)
  + Imap kliens könyvtár (pl.: JavaMail, Apache Commons Net)
  + Multithreading
* **Imap-downloader node**:
  + Docker környezet
* **Adatbázis kiszolgáló**:
  + Tetszőleges relációs adatbázis kezelő (ami rendelkezik JDBC driver-rel, pl: PostgreSQL, MySQL)
* **IMAP kiszolgáló**:
  + Tetszőleges IMAP kiszolgáló (pl.: Dovecot)

# Architektúra

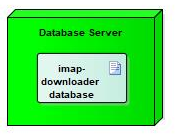
## IMAP kiszolgáló

Az rendszerben szükséges egy, vagy több IMAP kiszolgáló jelenléte. Minden IMAP kiszolgáló biztosítja egy, vagy több email fiók elérhetőségét.



## Adatbázis kiszolgáló

Egy relációs adatbázis kiszolgáló biztosítja az *imap-downloader* alkalmazás példányok számára a közös adatok elérhetőségét. Az adatbázis kezelő feladat biztosítani az egyes futó alkalmazás példányok között megosztott adatok szinkronizációját.

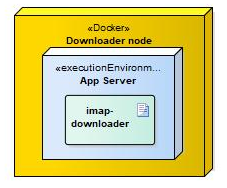


## Alkalmazás példány (node)

Az *imap-downloader* alkalmazás egy futó példánya egy önálló kiszolgálón, önálló *node* formájában fut. Az egyes futó példányok egymástól függetlenül működnek, egy node kiesése nem befolyásolja a többi példány működését.

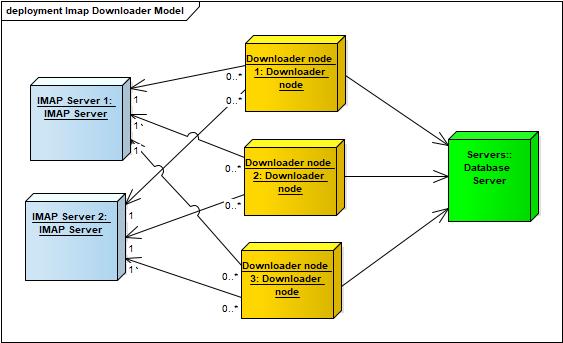
Az *imap-downloader* alkalmazás egy Docker image formájában kerül kiadásra, mely Docker image tetszőleges Docker-t használó környezetben egységesen futtatható.

Minden alkalmazás több szálon képes szinkronizációkat futtatni, egymással párhuzamosan, de a párhuzamos szinkronizációk egymástól függetlenül működnek, nem befolyásolják egymás működését.



## Rendszer felépítése

Egy működő rendszer az alábbi ábra szerint épül fel.



Egy rendszerben több IMAP kiszolgáló is lehet, melyek külön-külön is tetszőleges számú email fiókot szolgálhatnak ki.

Egy rendszerben tetszőleges számú IMAP letöltő alkalmazás példányt futtató node lehet. Egy letöltő alkalmazás példány egy feldolgozó szála bármelyik IMAP kiszolgálóhoz, és azon bármelyik email fiókhoz kapcsolódhat.

Egy rendszerben szükséges lennie egy központi adatbázis kiszolgálónak, amit megosztottan használnak az IMAP letöltő alkalmazás példányok.

# Működés

## Feldolgozás működése

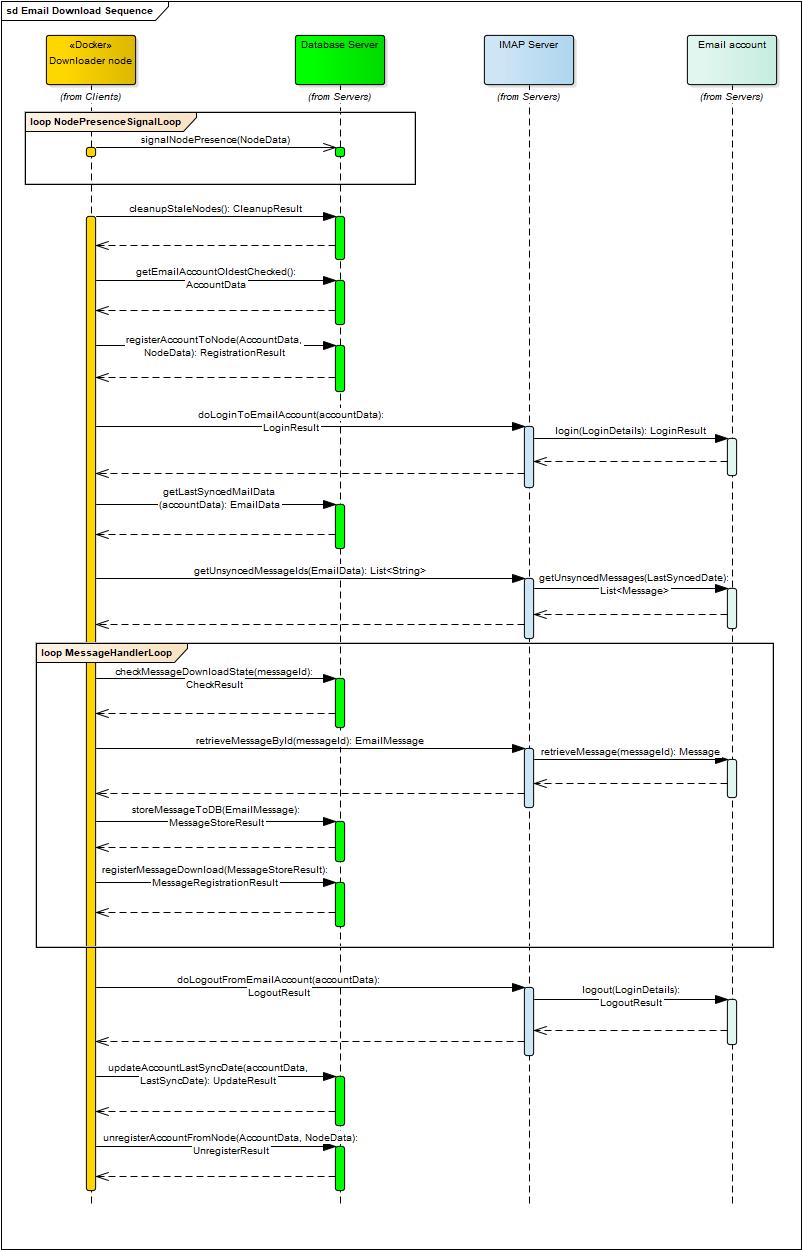
Egy email fiók feldolgozása minden esetben egy előre meghatározott algoritmus szerint (lásd.: [Szinkronizációs folyamat](#_Szinkronizációs_folyamat) fejezet), szigorúan meghatározott rend alapján történik.

Egy fiók szinkronizációja minden esetben egy önálló task formájában történik, ahol a task a [Szinkronizációs folyamat](#_Szinkronizációs_folyamat) fejezetben leírt algoritmust hajtja végre. A szinkronizációs task-ot egy adott node egy adott feldolgozó szála hajtja végre.

Amennyiben egy szinkronizációs task futása során hiba lép fel, úgy az adott task helyett egy új szinkronizációs task kerül futtatásra.

## Szinkronizációs folyamat

A szinkronizációs algoritmus a következő ábra szerinti lépéseket hajtja végre egy email fiókból történő adat letöltés során:



A szinkronizációs folyamat az egyes lépések során a következő tevékenységeket hajtja végre:

1. **NodePresenceSignalLoop**: Periodikusan, a node teljes futása során egy háttérszálban regisztrálja, hogy az adott node működik. Az adatbázisban regisztrálásra kerül, hogy az adott node utolsó jelzése mikor történt.
   * *signalNodePresence()* – Bejegyzi az adatbázisban, hogy az adott node az adott időpillanatban elérhető, működik.
2. *cleanupStateNodes()* – Ellenőrzi az adatbázisban szereplő node-ok utolsó frissítési időpontjait, és ami egy megadott határértéknél régebb óta nem jelzett, azt törli a regisztrációs adatbázisból. A törléssel együtt törlésre kerül minden, az adott node-hoz hozzárendelt email fiók bejegyzés is.
3. *getEmailAccountOldestChecked()* – Lekérdezi az adatbázisból a legrégebben feldolgozásra került, más node által nem regisztrált email fiók adatait.
4. *registerAccountToNode()* – Beregisztrálja, hogy a választott email fiókon az adott node megkezdi a feldolgozást. A sikeres regisztráció biztosítja, hogy más node ne kezdhessen szinkronizációt az adott email fiókon.
5. *doLoginToEmailAccount()* – Bejelentkezik az IMAP kiszolgálón keresztül a megadott email fiókba a feldolgozó folyamat.
6. *getLastSyncedMailData()* – Lekérdezi az adatbázisból az adott email fiókhoz kapcsolódóan utolsóként szinkronizált email üzenet adatait.
7. *getUnsyncedMessageIds()* – Lekérdezi az IMAP kiszolgálótól azon email üzenetek egyedi azonosítóit, melyek még nem kerültek szinkronizációra. A nem szinkronizált üzenetek meghatározásához az utolsóként szinkronizált email üzenet adatait használja fel.
8. **MessageHandlerLoop**: A nem szinkronizált üzenetek egyedi azonosítóit feldolgozó ciklus. A ciklus minden lépésében egy email üzenet kerül feldolgozásra.
   * *checkMessageDownloadState()* – Ellenőrzi az adatbázisban, hogy letöltésre került-e már korábban az üzenet.
   * *retrieveMessageById()* – Lekérdezi az IMAP kiszolgálótól az üzenet egyedi azonosítójával a teljes email üzenetet.
   * *storeMessageToDB()* – Az email üzenetet betárolja az adatbázisba.
   * *registerMessageDownload()* – A letöltött email üzenet sikeres letöltését és adatbázisban történő betárolását követően az email feldolgozása beregisztrálásra kerül az adatbázisban, ezzel biztosítva, hogy ne kerüljön újra letöltésre.
9. *doLogoutFromEmailAccount()* – Kijelentkezik az IMAP kiszolgálón keresztül a megadott email fiókból.
10. *updateAccountLastSyncDate()* – Frissíti az email fiók utolsó szinkronizációjának dátumát.
11. *unregisterAccountFromNode()* – Törli a feldolgozó node regisztrációját az adott email fiókhoz, így az újra feldolgozhatóvá kerül egy másik node által.

## Egyedi feldolgozás garantálása

A rendszernek garantálnia kell, hogy egy email fiók egy időpontban kizárólag egyetlen feldolgozó által kerüljön feldolgozásra. Ezt a feldolgozó node és az email fiók összerendelésével lehetséges megoldani.

A feldolgozó node-nak egy email fiók feldolgozásának megkezdése előtt szükséges regisztrálnia magát, mint a fiókot feldolgozó node. A regisztráció egy adatbázis bejegyzéssel történik. Amennyiben az adott fiókra van már másik node feldolgozó szálától regisztrációs bejegyzés, úgy az adott node nem kezdheti meg a fiók feldolgozását és másik fiókot szükséges választania.

Az adatbázisban a regisztrációs adatok konzisztenciájának biztosításához szükséges a regisztrációs adatokat tároló adattáblán megfelelő Unique kulcs létrehozása a fiók azonosítóból képezve, ami biztosítja, hogy egy adott fiók ne szerepelhessen több rekordban, ezzel biztosítva az egyszeri node-fiók regisztrációt.

## Node állapot kezelés

Minden node a futása során periodikusan, meghatározott időközönként bejegyez egy, az állapotát tartalmazó rekordot az adatbázisba, ami tartalmazza azt az időpontot, amikor az állapotjelentés küldésre került.

Egy node feldolgozó szála egy fiók feldolgozásának megkezdésekor ellenőrzi a(z aktív) node-ok állapotát tartalmazó tábláját, és törli azoknak a node-oknak a bejegyzéseit, melyek egy megadott határértéknél régebben jelezték az állapotukat. Ezek a node-ok innentől kezdve nem tekinthetők aktívnak addig, amíg újra nem küldenek friss állapotjelentést magukról.

Az inaktívvá vált node-ok feldolgozása során az inaktívvá vált node-okhoz regisztrált, feldolgozott fiókok is feloldásra kerülnek, a node-ok és a fiókok összekapcsolását tartalmazó adatbázis bejegyzések is törlésre kerülnek.

Amennyiben egy node fut, de nem tudja elvégezni a saját regisztrációját az adatbázisban (pl.: hálózati hiba miatt a node és az adatbázis között), úgy a node szükséges, hogy felfüggessze a saját működését addig, amíg a kapcsolat helyre nem áll a node és az adatbázis között.

## Letöltött üzenetek kezelése

Egy node által egy fiókból letöltött üzenet minden esetben a következő feldolgozási lépéseken megy át:

1. Üzenet letöltése az IMAP kiszolgálóról.
2. A letöltött üzenet adatai betárolásra kerülnek az adatbázisba.
3. Az üzenet letöltése regisztrálásra kerül az adatbázisba. Ez a regisztráció biztosítja, hogy egy üzenet ne kerüljön újra letöltésre, ha feldolgozási hiba lépett fel egy node esetén az adott fiók feldolgozása során, és egy másik node-nak újra fel kell dolgoznia az üzeneteket.

## Fiókok feldolgozási sorrendjének kezelése

A fiókok feldolgozása során szükséges kivédeni, hogy egy fiók túl hosszú ideig ne kerüljön feldolgozásra. Ehhez minden node egy adott fiók feldolgozásának lezárása részeként bejegyzi az adatbázisba, hogy az adott fiókot mikor szinkronizálta. Amikor egy node lekérdezi a feldolgozható fiókok listáját (azon fiókok, melyekhez nincs beregisztrálva aktuálisan feldolgozó node), azt minden esetben az utolsó szinkronizáció időpontja szerinti növekvő sorrendben teszi. Így a node által feldolgozásra kapott fiók mindig a legrégebben szinkronizált fiók lesz.

Elkerülendő a fiókok túl gyakori ellenőrzését (pl.: ha egyik fiókban sincs feldolgozható üzenet, és a feldolgozó task-ok gyorsan végeznek), célszerű a feldolgozható fiókok lekérdezésébe bevezetni egy plusz feltételt, mely meghatároz egy minimum időintervallumot aminek legalább el kell telnie (pl.: 1 perc), mielőtt egy fiók újra feldolgozásra kiadásra kerülhet.

# Továbbfejlesztési lehetőségek

A leírt rendszer a következő potenciális lehetőségekkel lehet módosítható, bővíthető:

* **Adatbázis cluster-ezés**: egy adatbázis cluster beállításával növelhető az adatbázis teljesítménye, valamint biztosítható az adatbázis redundanciája, és ezzel biztosítható a nagyobb rendelkezésre állása.
* **Köztes REST api illesztése az adatbázis és a node-ok közé**: A node-ok közvetlen adatbázis kapcsolata kiváltható egy köztes REST interface rendszerbe illesztésével. Ez az köztes interface kapcsolódna az adatbázishoz, illetve REST-en keresztül fogadná a node-ok kéréseit.  
  Az új köztes interface illesztése esetén figyelembe kell venni, hogy egyetlen interface példány rendszerbe illesztése esetén az egyetlen példány mint „single point of failure” viselkedik, kiesése esetén a teljes rendszer működése leáll. Emiatt célszerű lehet valamilyen *emergency backup*, vagy elosztott megoldás alkalmazása az interface példányra.