Specifica Tecnica

2024-03-18 — v0.2.0

overture.unipd@gmail.com

Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin *Zextras* Gruppo *Overture*
Responsabile	Francesco Costantino Bulychov
Redattori	Eleonora Amadori Michele Bettin Riccardo Bonavigo Francesco Costantino Bulychov Riccardo Fabbian Francesco Furno Alex Vedovato
Verificatori	Eleonora Amadori Michele Bettin Riccardo Bonavigo Francesco Costantino Bulychov Riccardo Fabbian Francesco Furno Alex Vedovato

Registro delle modifiche

Versione	Data	Autori	Verificatori	Dettaglio
0.2.0	2024-03-18	Alex Vedovato	Michele Bettin	Aggiornato lo stato dei requisiti funzionali.
0.1.1	2024-03-13	Eleonora Amadori	Riccardo Bonavigo	Aggiunta la sezione ‘API’.
0.1.0	2024-03-13	Alex Vedovato	Riccardo Fabbian	Migliorata la sezione ‘Diagramma delle classi’.
0.0.21	2024-03-12	Michele Bettin	Francesco Furno	Migliorata la sezione ‘Database’.
0.0.20	2024-03-10	Francesco Furno	Riccardo Bonavigo	Migliorata la sezione ‘Architettura logica’.
0.0.19	2024-03-09	Francesco Costantino Bulychov	Michele Bettin	Aggiunto ‘Adapter’ all’interno della sezione ‘Design pattern utilizzati’.
0.0.18	2024-03-08	Eleonora Amadori	Alex Vedovato	Integrate le tecnologie mancanti.
0.0.17	2024-03-06	Francesco Furno	Francesco Costantino Bulychov	Modifiche dovute alla risoluzione dei problemi rilevati nel colloquio con il Prof. Riccardo Cardin.
0.0.16	2024-03-05	Eleonora Amadori	Francesco Costantino Bulychov	Aggiunto ‘Builder’ all’interno della sezione ‘Design pattern utilizzati’.
0.0.15	2024-03-02	Michele Bettin	Eleonora Amadori	Rimosso ‘Chain of Responsibility’ dalla sezione ‘Design pattern utilizzati’.
0.0.14	2024-03-02	Riccardo Fabbian	Francesco Costantino Bulychov	Aggiunta la sottosezione ‘Interfacciamento al database’ all’interno della sezione ‘Diagramma delle classi’.
0.0.13	2024-03-01	Francesco Furno	Riccardo Bonavigo	Aggiunta la sottosezione ‘Gestione delle richieste’ all’interno della sezione ‘Diagramma delle classi’.
0.0.12	2024-02-29	Eleonora Amadori	Riccardo Bonavigo	Aggiunto ‘Chain of Responsibility’ all’interno della sezione ‘Design pattern utilizzati’.
0.0.11	2024-02-27	Riccardo Fabbian	Eleonora Amadori	Aggiunta la sottosezione ‘Ingresso delle richieste nell’applicazione’ all’interno della sezione ‘Diagramma delle classi’.
0.0.10	2024-02-26	Michele Bettin	Francesco Costantino Bulychov	Aggiunto ‘Dependecy injection’ all’interno della sezione ‘Design pattern utilizzati’.
0.0.9	2024-02-24	Eleonora Amadori	Riccardo Bonavigo	Aggiunta la sezione ‘Diagramma delle classi’.

0.0.8	2024-02-20	Francesco Costantino Bulychov	Francesco Furno	Aggiunta la sezione ‘Database’.
0.0.7	2024-02-20	Alex Vedovato	Riccardo Fabbian	Aggiunta la sezione ‘Architettura di deployment’.
0.0.6	2024-02-19	Riccardo Bonavigo, Riccardo Fabbian	Michele Bettin	Aggiunta la sezione ‘Architettura logica’.
0.0.5	2024-02-16	Riccardo Bonavigo	Michele Bettin	Aggiunta la sezione ‘Tecnologie per il testing’.
0.0.4	2024-02-16	Riccardo Fabbian	Francesco Furno	Aggiunta la sezione ‘Tecnologie per l’analisi del codice’.
0.0.3	2024-02-15	Francesco Costantino Bulychov	Riccardo Fabbian	Aggiunta la sezione ‘Tecnologie per la codifica’.
0.0.2	2024-02-15	Alex Vedovato	Francesco Furno	Aggiunta la sezione ‘Stato dei requisiti funzionali’.
0.0.1	2024-02-12	Riccardo Fabbian, Alex Vedovato	Michele Bettin	Struttura di base del documento e introduzione.

Indice

1) Introduzione 6
1.1) Scopo del documento 6
1.2) Glossario 6
1.3) Riferimenti 6
1.3.1) Riferimenti normativi 6
1.3.2) Riferimenti informativi 6
2) Tecnologie 8
2.1) Tecnologie per la codifica 8
2.1.1) Linguaggi 8
2.1.2) Strumenti e servizi 8
2.1.3) Framework 8
2.1.4) Librerie 9
2.2) Tecnologie per l'analisi del codice 10
2.2.1) Analisi statica 10
2.2.2) Analisi dinamica 10
2.3) Tecnologie per il testing 11
2.3.1) Linguaggi 11
2.3.2) Framework 11
3) API 12
3.1) Jmap (standard) 12
3.2) Jmap (custom) 12
3.2.1) GET 12
3.2.2) POST 12
3.3) Upload 13
3.4) Download 13
4) Architettura 14
4.1) Architettura logica 14
4.2) Architettura di deployment 15
4.2.1) Struttura a monolite vs struttura a microservizi 15
4.2.2) Docker e conteinerizzazione dell'applicazione 15
4.3) Design pattern utilizzati 18
4.3.1) Dependency injection 18
4.3.2) Adapter 20
4.3.3) Builder 22
4.4) Diagramma delle classi 25
4.4.1) Ingresso delle richieste nell'applicazione 26
4.4.2) Gestione delle richieste 31
4.4.3) Interfacciamento al database 36
4.5) Database 42
4.5.1) Scelta di RethinkDB 42
4.5.2) Funzionalità di RethinkDB 42
4.5.3) Utilizzo di RethinkDb nel nostro progetto 42
4.5.4) Utilizzo di MinIO 44
4.5.5) Conclusioni 45
5) Stato dei requisiti funzionali 46
5.1) Grafici riassuntivi 55

Lista della immagini

Figura 1: Architettura logica del prodotto 14
Figura 2: Architettura di deployment del prodotto 16
Figura 3: Diagramma delle classi 25
Figura 4: Modellazione delle componenti che gestiscono l'ingresso delle richieste nell'applicazione 26
Figura 5: Modellazione delle componenti che gestiscono le rischieste all'interno dell'applicazione . 31
Figura 6: Modellazione delle componenti che gestiscono l'interfacciamento al database 36
Figura 7: Stato dei requisiti funzionali totali 55
Figura 8: Stato dei requisiti funzionali obbligatori 55
Figura 9: Stato dei requisiti funzionali desiderabili 56
Figura 10: Stato dei requisiti funzionali opzionali 56

Lista delle tabelle

Tabella 1: Linguaggi di programmazione usati per la codifica 8
Tabella 2: Strumenti e servizi usati per la codifica 8
Tabella 3: Framework usati per la codifica 8
Tabella 4: Librerie usate per la codifica 9
Tabella 5: Tecnologie usate per l'analisi statica del codice 10
Tabella 6: Tecnologie usate per l'analisi dinamica del codice 10
Tabella 7: Linguaggi di programmazione usati per il testing 11
Tabella 8: Framework utilizzati per il testing 11
Tabella 9: Stato dei requisiti funzionali 46

1) Introduzione

1.1) Scopo del documento

Lo scopo di questo documento è quello di elencare e motivare le scelte architetturali che il gruppo Overture ha intrapreso per la realizzazione dell'infrastruttura informatica richiesta. Il documento comprende anche i diagrammi delle classi e dei package al fine di spiegare in maniera più chiara e dettagliata il software sviluppato.

1.2) Glossario

Per evitare ambiguitá o incomprensioni riguardanti la terminologia usata nel documento, é stato deciso di adottare un glossario in cui vengono riportate le varie definizioni. In questa maniera in esso verranno riportati tutti i termini specifici del dominio d'uso con relativi significati.
La presenza di un termine all'interno del `Glossario` viene indicata applicando *questo stile*.

1.3) Riferimenti

1.3.1) Riferimenti normativi

- `Norme di Progetto v1.0.0`:
  https://overture-unipd.github.io/docs/rtb/interni/norme_di_progetto_v1.0.0.pdf
- **PD2 - Regolamento del progetto didattico**
  https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf
- **Capitolato d'appalto C8**: JMAP, il nuovo protocollo standard per la comunicazione email
  https://www.math.unipd.it/~tullio/IS-1/2023/Progetto/C8.pdf

1.3.2) Riferimenti informativi

- `Glossario v1.0.0`:
  https://overture-unipd.github.io/docs/rtb/interni/glossario_v1.0.0.pdf
- `Analisi dei Requisiti v1.1.0`:
  https://overture-unipd.github.io/docs/rtb/esterni/analisi_dei_requisiti_v1.1.0.pdf
- Progettazione: le dipendenze fra le componenti, Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2023/Object-Oriented%20Progamming%20Principles%20Revised.pdf
- Progettazione e programmazione: diagrammi delle classi (UML), Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf
- Progettazione: i *pattern* architetturali, Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
- Progettazione: il *pattern Dependency Injection*, Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%20-%20Dependency%20Injection.pdf
- Progettazione *software* (T6), Prof. Tullio Vardanega
  https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/T6.pdf
- Progettazione: i *pattern* creazionali (GoF), Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
- Progettazione: i *pattern* strutturali (GoF), Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf
- Progettazione: i *pattern* di comportamento (GoF), Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2021/Design%20Pattern%20Comportamentali_4x4.pdf

- Programmazione: SOLID *programming*, Prof. Riccardo Cardin
  https://www.math.unipd.it/~rcardin/swea/2021/SOLID%20Principles%20of%20Object-Oriented%20Design_4x4.pdf

2) Tecnologie

In questa sezione vengono elencate tutte le tecnologie utilizzate per l'implementazione del prodotto richiesto dal capitolato.

2.1) Tecnologie per la codifica

2.1.1) Linguaggi

Nome	Versione	Descrizione
Java	21 LTS	Linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica. Fortemente consigliato dall'azienda proponente Zextras dato che è il linguaggio principale nello stack tencologico di Carbonio.

Tabella 1: Linguaggi di programmazione usati per la codifica

2.1.2) Strumenti e servizi

Nome	Versione	Descrizione
Docker	25.0.0	Strumento di conteinerizzazione richiesto esplicitamente dall'azienda proponente. Ogni contenitore viene eseguito in modo isolato dagli altri. Si integra facilmente con Gradle tramite un plugin.
Docker-compose	2.24.0	Strumento di conteinerizzazione utilizzato per la definizione di servizi multi-container. Semplifica la gestione e la scalabilità delle applicazioni.
RethinkDB	2.4.4	Database NoSQL che supporta subqueries e changefeed. Facilita la gestione e le conversioni degli oggetti Json e consente di rappresentare facilmente dati complessi e annidati.
Gradle	8.6	Strumento di automazione della build e di gestione delle dipendenze, progettato per la compilazione, il testing e la distribuzione di progetti software. Utilizza un linguaggio DSL basato su Groovy o Kotlin e supporta la build incrementale.
Caddy	2.7.0	Web server open-source che converte il traffico da HTTP a HTTPS. Si distingue per la sua facilità d'uso, configurazione automatizzata e supporto nativo per HTTPS. Necessario in quanto richiesto dai client.
MinIO	Release 2024-01-31	Server open-source progettato per implementare l'archiviazione di oggetti in modo scalabile e distribuito.

Tabella 2: Strumenti e servizi usati per la codifica

2.1.3) Framework

Nome	Versione	Descrizione
Spark	2.9.4	Lightweight framework open-source per lo sviluppo di applicazioni web in Java. Ideale per progetti agili che cercano di mantenere la complessità sotto controllo.
Guice	7.0.0	Lightweight framework open-source per Java progettato per semplificare lo sviluppo di applicazioni attraverso l'iniezione di dipendenze. Offre un modo semplice e dichiarativo per gestire le dipendenze tra i componenti che costituiscono l'applicazione, migliorando la manutenibilità e la testabilità del codice. Ideale per progetti che vogliono mantenere la complessità sotto controllo.

Tabella 3: Framework usati per la codifica

2.1.4) Librerie

Nome	Versione	Descrizione
Java JMAP iNPUTmice Library	0.8.18	Libreria consgliata dall'azienda proponente che sincronizza i dati tra client e server utilizzando il JSON Meta Application Protocol. Definisce classi per ogni operazione possibile, funzioni per operare su di esse e conversioni da e verso Json.

Tabella 4: Librerie usate per la codifica

2.2) Tecnologie per l'analisi del codice

2.2.1) Analisi statica

Nome	Versione	Descrizione
Compilatore Java	JDK 21	Traduce il codice sorgente scritto in linguaggio di programmazione Java in un formato eseguibile, generalmente chiamato bytecode Java. Il bytecode Java può essere eseguito da una macchina virtuale Java (JVM) su qualsiasi piattaforma che abbia una JVM disponibile, rendendo il codice Java altamente portabile.
Spotless	6.23.3	Plugin Gradle open-source utilizzato per applicare automaticamente le convenzioni di formattazione del codice a progetti Java. Garantisce che il codice sorgente sia formattato secondo regole specifiche, migliorandone la leggibilità e la manutenibilità.

Tabella 5: Tecnologie usate per l'analisi statica del codice

2.2.2) Analisi dinamica

Nome	Versione	Descrizione
Junit	5.9.3	Framework open source per l'automazione di unit testing per il linguaggio di programmazione Java. Offre un ambiente di sviluppo strutturato per la scrittura dei test, consentendo agli sviluppatori di verificare il comporamento delle singole unità di codice in modo efficiente e affidabile.
Mockito	5.1.1	Framework open surce di testing per il linguaggio di programmazione Java. È utlizzato per la creazione, configurazione e gestione di oggetti mock nei test di unità. Consente agli sviluppatori di simulare il comporamento di oggetti reali durante l'esecuzione dei test.
TestContainers	1.19.3	Framework open source che fornisce un'interfaccia per l'integrazione di container Docker nelle attività di testing. È particolarmente utile per lo sviluppo di test di integrazione che coinvolgono componenti dipendenti come database o qualsiasi altro servizio che può essere eseguito in un container Docker.

Tabella 6: Tecnologie usate per l'analisi dinamica del codice

2.3) Tecnologie per il testing

2.3.1) Linguaggi

Nome	Versione	Descrizione
Python	3.12.1	Linguaggio di programmazione ad alto livello, interpretato e general-purpose. È noto per la sua sintassi chiara e leggibile, che lo rende molto adatto per sviluppare rapidamente script e applicazioni.

Tabella 7: Linguaggi di programmazione usati per il testing

2.3.2) Framework

Nome	Versione	Descrizione
Locust	2.23.1	Framework open source di testing di carico e stress delle applicazioni. È scritto in Python e permette agli sviluppatori di scrivere test di carico simulando il comporamento di migliaia di utenti concorrenti.
Postman	10.22	Strumento di collaborazione per lo sviluppo di API. In particolare ne semplifica i processi di sviluppo, test e documentazione. Consente la creazione di richieste HTTP personalizzate.

Tabella 8: Framework utilizzati per il testing

3) API

Nel contesto del nostro progetto, le API costituiscono il principale punto di accesso attraverso il quale i client possono interagire con il sistema.
Di seguito vengono esaminate le diverse API fornite all'interno del nostro prodotto; ciascuna di esse è soggetta a verifica attraverso il filtro di autenticazione `authenticate`, `/api/*`, che viene eseguito prima di procedere con l'effettiva gestione della richiesta.

3.1) Jmap (standard)

Questo endpoint è standard ed è definito all'interno della specifica di JMAP. Esegue il redirect all'endpoint JMAP da noi definito.
- **Endpoint:** `/.well-known/jmap`;
- **Metodo HTTP:** `GET`;
- **Ritorno**: `null`.

3.2) Jmap (custom)

Questo endpoint custom funziona in due modi: `GET` e `POST`, entrambi autenticati.

3.2.1) GET

Il metodo `GET` consente di ricevere l'oggetto `session` definito dallo standard.
- **Endpoint:** `/api/jmap`;
- **Metodo HTTP:** `GET`;
- **Ritorno**: oggetto `session`.

Esito	Codice HTTP	Descrizione
Positivo	**200**: OK	L'oggetto session viene inviato correttamente al client.
Negativo	**401**: Unauthorized	Le credenziali inserite sono errate.
Negativo	**500**: Internal Server Error	Si è verificato un errore imprevisto durante la ricezione dell'oggetto `session`.

3.2.2) POST

Il metodo `POST` consente di inoltrare "metodi" JMAP, così definiti dallo standard.
- **Endpoint:** `/api/jmap`;
- **Metodo HTTP:** `POST`;
- **Ritorno**: risposta dei metodi JMAP inviati.

Esito	Codice HTTP	Descrizione
Positivo	**200**: OK	La richiesta contenente uno o più metodi JMAP viene inviata correttamente al server.
Negativo	**401**: Unauthorized	Le credenziali inserite sono errate.
Negativo	**500**: Internal Server Error	Si è verificato un errore imprevisto durante l'invio della richiesta.

3.3) Upload

Upload è un endpoint di tipo di tipo REST che consente di caricare un allegato dal database MinIO. Rappresenta una richiesta autenticata.

- **Endpoint:** `/api/upload`;
- **Metodo HTTP:** `GET`;
- **Ritorno**: stringa JSON che rappresenta l'oggetto appena caricato.

Esito	Codice HTTP	Descrizione
Positivo	**200**: OK	L'upload ha successo e i dati caricati sono stati inviati correttamente al server.
Negativo	**401**: Unauthorized	Le credenziali inserite sono errate.
Negativo	**500**: Internal Server Error	Si è verificato un errore imprevisto durante l'upload.

3.4) Download

Download è un endpoint di tipo di tipo REST che consente di scaricare un allegato dal database MinIO. Rappresenta una richiesta autenticata.

- **Endpoint:** `/api/download`;
- **Metodo HTTP:** `GET`;
- **Ritorno**: octet stream (byte) che rappresentano l'oggetto appena scaricato.

Esito	Codice HTTP	Descrizione
Positivo	**200**: OK	Il download ha successo e i dati richiesti sono stati inviati correttamente al client.
Negativo	**401**: Unauthorized	Le credenziali inserite sono errate.
Negativo	**404**: Not found	I dati richiesti dal client non sono stati trovati nel server.
Negativo	**500**: Internal Server Error	Si è verificato un errore imprevisto durante il download.

4) Architettura

La descrizione dell'architettura del prodotto adotta un approccio top-down, partendo dalla struttura generale per poi scendere nel dettaglio.

4.1) Architettura logica

Nell'architettura logica che abbiamo scelto di adottare, il server di posta è organizzato in un modello esagonale che riflette una suddivisione chiara delle responsabilità e delle funzionalità delle varie componenti, andando a porre al centro la business logic, la quale non andrà così a dipendere da altre parti del sistema riguardanti, per esempio, logiche di persistenza.

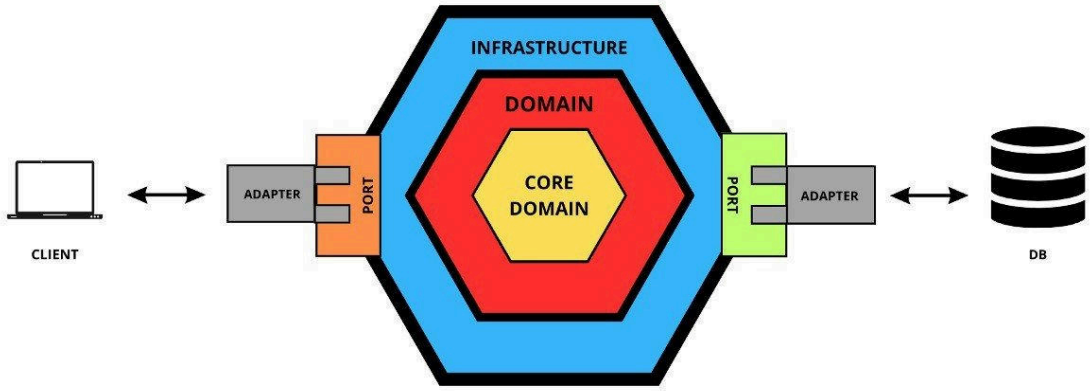

Figura 1: Architettura logica del prodotto

All'esterno dell'esagono, in entrata, è presente una componente dedicata alla gestione delle richieste provenienti dai client, fornendo un'interfaccia per l'ingresso di quest'ultime nel sistema. Questa classe è responsabile di definire molteplici rotte utilizzando il framework Spark per poi far entrare nell'infrastruttura le richieste attraverso specifiche porte. È il punto di contatto tra il server di posta elettronica e il mondo esterno, quindi i client che interagiscono con il nostro prodotto.

A questo punto i vari controller andranno a effettuare i controlli relativi all'application logic del nostro prodotto, garantendo che le richieste siano conformi allo standard, eseguendo una prima validazione dei dati. Nel caso in cui la libreria fallisse la conversione da JSON a oggetto, una determinata richiesta non verrà portata a termine prima ancora di raggiungere la business logic del prodotto.

Al centro dell'esagono risiede poi la business logic del server di posta elettronica. Questa è la parte centrale del sistema, dove avviene l'elaborazione delle richieste in arrivo. Qui si trovano quindi le implementazioni delle funzionalità come la gestione delle email o la gestione delle caselle di posta, oltre a tutti gli oggetti della libreria specifici del dominio del nostro prodotto. La business logic costituisce il cuore del server di posta elettronica, garantendo il corretto funzionamento del sistema e la coerenza delle operazioni svolte.

Infine, all'esterno dell'esagono, in uscita, troviamo l'insieme di componenenti che si occupano dell'interfacciamento con il database. Queste classi hanno il ruolo di gestire la persistenza dei dati necessari per il funzionamento del sistema, inclusi salvataggio delle email, memorizzazione delle cartelle, persistenza degli account e così via. Assicurano che i dati vengano salvati e recuperati in modo affidabile ed efficiente.

L'adozione di questa architettura esagonale favorisce una gestione modulare e scalabile del server di posta elettronica. Ogni componente svolge un ruolo specifico e ben definito, facilitando la manutenzione, l'aggiornamento e l'espansione del sistema nel tempo. Inoltre, la chiara separazione delle re-

sponsabilità e delle funzionalità promuove la testabilità del sistema, consentendo una maggiore fiducia nella robustezza e nella stabilità complessiva del prodotto.

Un grande vantaggio di questa architettura è la facilità nel cambiare, per esempio, il database sottostante. Supponendo infatti che si voglia aggiornare il sistema di persistenza dei dati, grazie alla netta separazione delle componenti e all'interfacciamento ben definito con il database è relativamente semplice farlo senza dover apportare modifiche significative al resto del sistema. Questa flessibilità consente di adattare il server di posta elettronica alle esigenze future e alle evoluzioni tecnologiche, garantendo una maggiore longevità e versatilità del prodotto.

4.2) Architettura di deployment

4.2.1) Struttura a monolite vs struttura a microservizi

Dal momento in cui il prodotto software che dobbiamo andare a realizzare ha lo scopo di fornire una implementazione per un nuovo protocollo, per poi successivamente testarne le performance con degli stress test, la scelta di un architettura a monolite rispetto che ad altre, come quella a microservizi, è motivata da una vasta gamma di fattori.
Prima di tutto, come detto prima, il software non necessiterà di particolari espansioni future, una volta forniti i risultati degli stress test essa verra utilizzata dal committente per effettuare ulteriori test o al massimo per osservare come abbiamo implementato certe funzionalità di jmap nel caso volessero integrarle nei loro sistemi.
Fatte queste premesse, l'approccio monolitico è quindi preferito per la sua rapidità e semplicità. L'applicazione è destinata ad essere di dimensioni limitate e a svolgere un compito specifico, il team dunque intende semplificare lo sviluppo senza la necessità di gestire la complessità aggiuntiva introdotta da un'architettura a microservizi.
Inoltre, in un progetto dove il team di sviluppo ha un esperienza di programmazione scarsa e necessita di modificare e correggere continuamente alcune parti del software, l'architettura monolitica risulta essere più gestibile in quanto permette di semplificare il processo di sviluppo riducendo la necessità di coordinazione tra diversi servizi, come richiesto dalle architetture a microservizi.

4.2.1.1) Conclusioni

L'architettura monolitica è raccomandata per applicazioni semplici e per prototipi, semplificando lo sviluppo senza la necessità di integrare molteplici servizi. D'altra parte l'architettura a microservizi si adatta meglio a sistemi complessi offrendo un'aggiunta flessibile di nuove funzionalità.
Per la scelta dell'architettura è anche fondamentale conoscere le competenze generali del team di sviluppo. Per lo sviluppo con i microservizi sono generalmente essenziali conoscenze tecniche specifiche riguardanti cloud, API e contenerizzazione.
In secondo luogo la scelta dell'architettura è anche condizionata dall'infrastruttura con cui si ha a che fare: le applicazioni monolitiche operano su un singolo server, i microservizi traggono maggiore beneficio dall'ambiente cloud, offrendo scalabilità e distribuzione.
Per conludere, il team *Overture* riconosce i vantaggi che alcune architetture più complesse, come quella a microservizi, porterebbero sicuramente al prodotto sviluppato, ma per una serie di ragioni legate alla complessità instriseca delle architetture stesse riteniamo che nel nostro scenario i vantaggi non coprano assolutamente gli sforzi e i tempi necessari nel presente per adottare queste architetture. Per questo abbiamo optato per una architettura monolitica.

4.2.2) Docker e conteinerizzazione dell'applicazione

L'uso della conteinerizzazione nell'ambito dell'architettura di deployment è considerato interessante per i seguenti motivi (i quali ci hanno fatto comprendere il motivo per cui il comittente aveva inserito la containerizzazione dell'applicazione come requisito obbligatorio):

- Portabilità: i container forniscono un ambiente isolato che include tutte le dipendenze. Questo rende l'applicazione altamente portabile tra diversi ambienti, eliminado le preoccupazioni legate alle differenze di configurazione tra i sistemi di sviluppo e produzione.
- Consistenza: la containerizzazione garantisce che l'applicazione venga eseguita in un ambiente consistente, indipendentemente dalla macchina host. Ciò riduce i problemi legati a differenze di configurazione tra le diverse fasi di sviluppo e deployment, migliorando la coerenza del processo.
- Scalabilità: i container sono leggeri e possono essere avviati rapidamente. Questo facilita la scalabilità orizzontale, consentendo l'esecuzione di più istanze di un'applicazione su più container.
- Gestione delle risorse: i container condividono il kernel del sistema operativo host, riducendo l'overhead rispetto a soluzioni di virtualizzazione tradizionali. Ciò consente di utilizzare in modo più efficiente le risorse hardware, riducendo i costi e migliorando le prestazioni complessive del sistema.

Complessivamente, la containerizzazione offre numerosi vantaggi nell'ambito dell'architettura di deployment, migliorando l'efficienza, la portabilità e la gestione delle applicazioni.

4.2.2.1) Come lo abbiamo utilizzato noi

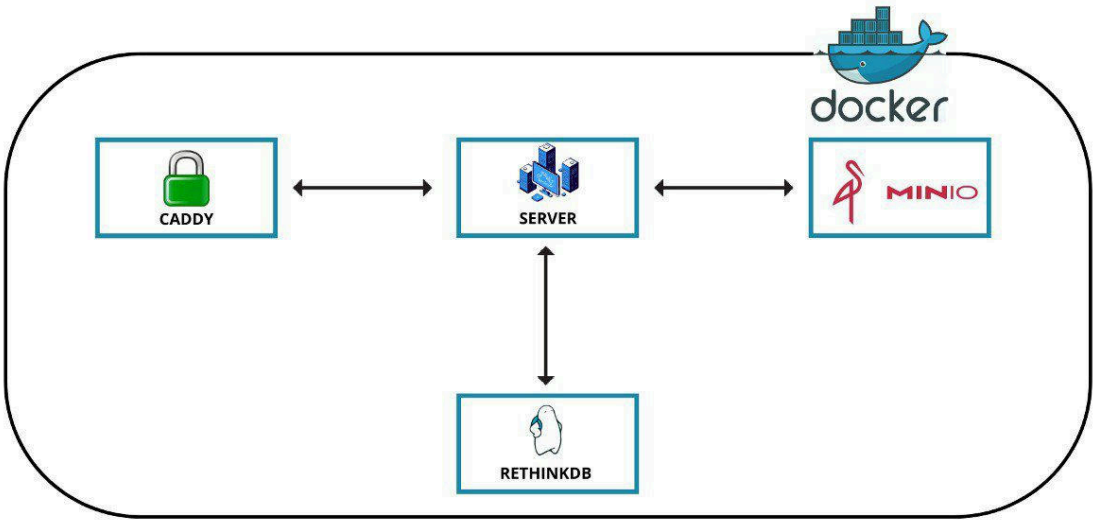

Figura 2: Architettura di deployment del prodotto

Se andiamo sulla cartella root del nostro progetto noteremo che c'è un file chiamato "docker-compose.yml". Questo definisce una configurazione di Docker Compose per orchestrare i container di tre servizi diversi: web server, database e caddy.
Il servizio web server è attivato mediante l'immagine custom `overture-unipd/jmap:latest` ricavata dall'immagine di base `openjdk:21-jdk-slim` e esposta nella porta 8000.
Il servizio di database è creato mediante l'immagine `rethinkdb:2.4.2-bullseye-slim` disponibile direttamente sul repository pubblico al seguente link https://hub.docker.com/_/rethinkdb mappando le porte 9000, 29015 e 28015 del container con rispettivamente le porte 8080. 29015 e 28015 dell'host e configurando i volumi in modo tale da consentire la persistenza dei dati del database tra le esecuzioni del container.
Il servizio di caddy lo attiviamo con l'immagine custom `overture-unipd/caddy:latest` ricavata dall'immagine di base `caddy:latest` ma integrata con il plugin per Duck DNS. Anche qui vengono mappate le porte 80 e 443 del container con quelle dell'host e vengono configurati i volumi Docker per condividere dati tra il container e l'host, ad esempio per persistere i dati del server web Caddy e per

fornire un file di configurazione Caddy personalizzato.
Il servizio MinIO è creato mediante l'immagine `minio/minio:RELEASE.2024-02-24T17-11-14Z` disponibile direttamente sul repository pubblico di Docker Hub al seguente link: https://hub.docker.com/r/minio/minio. Vengono mappate le porte: 100000 e 100001 con le rispettive porte: 9000 e 9001. Il volume minio:/data viene mappato per memorizzare i dati di Minio nella directory /data all'interno del contenitore. Questo significa che i dati di Minio saranno memorizzati in modo persistente nel volume Docker, consentendo al contenitore di accedervi e manipolarli anche dopo il riavvio.

Si nota che tutti i volumi che i servizi montano, sono riportati alla fine del file "docker-compose.yml" dopo il record `volumes`.

4.3) Design pattern utilizzati

4.3.1) Dependency injection

4.3.1.1) Motivazioni e studio del design pattern

In qualsiasi progetto non banale, quindi costituito da un numero di componenti considerevole, risulta fondamentale la gestione e quindi la minimizzazione delle dipendenze. Questo non tanto nel momento in cui si sviluppa il codice, ma più nella fase di manutenzione o implementazione di nuove features nel momento in cui il software è gia in produzione. Lo scopo di questo design pattern è quindi quello di separare il comportamento di un componente dalla risoluzione delle sue dipendenze con l'obiettivo di semplificare l'albero delle dipendenze. Se tale albero diventa troppo complesso (aggiungendo debito tecnico) si ha una situazione in cui ogni componente del progetto dipende da altri, facendo si che una modifica a quest'ultimo comporti delle modifiche a cascata di tutti gli altri componenti.

4.3.1.2) Implementazione del design pattern

Il gruppo di progetto ha deciso di implementare il pattern dependecy injection con il framework Guice. Semplicemente, Guice ci permette di realizzare questo design pattern evitando di scrivere tutto il boilerplate code necessario se lo implementassimo con il metodo tradizionale senza l'utilizzo di strumenti esterni. In parole povere, Guice allevia la necessità di avere componenti factories nel proprio codice e di usare la parola chiave `new` per ogni oggetto che si vuole costruire. Il costrutto @Inject di Guice è il nuovo `new` di Java. Bisognerà comunque implementare delle classi factories in certi casi, ma il codice non dipenderà direttamente da esse e il codice risultante sarà più facile da modificare, da testare e da riutilizzare.

4.3.1.2.1) Concetti principali di Guice ed esempio di utilizzo

Dependecy Injection è un pattern dove le classi dichiarano le loro dipendenze come argomenti al posto di creare oggetti legati a queste dipendenze direttamente al loro interno. Ad esempio, un client che vorrebbe chiamare un servizio non dovrebbe sapere come costruire questo servizio, ma al contrario, del codice esterno deve essere responsabile per fornire questo servizio al client.

4.3.1.2.1.1) Costruttore @Inject

Tutte le classi di Java che sono annotate con @Inject possono essere chiamate da Guice tramite un processo di "costructor injection", dove ogni argomento viene creato e fornito da Guice stesso. Questo è un esempio di classe demo che usa il "costructor injection":

```
class Saluto {
  private final String messaggio;
  private final int conta;
```

```
  // La classe Saluto dichiara che necessita di una stringa per il messaggio
  // e un intero che rappresenta il numero di volte che un messaggio è
  // stampato.
```

```
  // L'annotazione @Inject segna il costruttore come istanziabile da Guice.
  @Inject
  Saluto(@Messaggio String messaggio, @Conta int conta) {
    this.messaggio = messaggio;
    this.conta = conta;
  }

  ...
}
```

La classe Saluto ha un costruttore che è chiamato quando l'applicazione chiede a Guice di istanziare un oggetto Saluto. Guice creerà i due argomenti richiesti e invocherà il costruttore. Le dipendeze, che sono gli argomenti da passare al costruttore, sono noti a Guice grazione ai codiddetti: Moduli, che soddisfano queste dipendenze.

4.3.1.2.1.2) Moduli di Guice

I moduli sono dei costrutti di Guice che permettono di soddisfare le dipendenze richieste dal costruttore di una classe. Questo è fatto grazie alla creazione di una classe (il modulo appunto) che specifica come soddisfare tali dipendenze automaticamente. Ecco un esempio di modulo che soddisfa le dipendenze della classe Saluto:

```
import com.google.inject.Provides;

class DemoModulo extends AbstractModule {
  @Provides
  @Conta
  static Integer fornisciConta() {
    return 3;
  }

  @Provides
  @Messaggio
  static String fornisciMessaggio() {
    return "hello world";
  }
}
```

4.3.1.2.1.3) Utilizzo

Incapsulando il codice precedente in una classe apposita di demo ecco come possiamo usare l'infrastruttura creata per testare la potenza di Guice:

```
public static void main(String[] args) {
    /*
     * Guice.createInjector() prende uno o più moduli e ritorna una nuova istanza di
Inject.
     */
    Injector injector = Guice.createInjector(new DemoModulo());

    /*
     * Ora che abbiamo l'injector possiamo creare un istanza della classe Saluto.
     */
    Saluto saluto = injector.getInstance(Saluto.class);

    // Stampa "hello world" 3 volte nella console.
    greeter.diciCiao();
  }
```

4.3.1.3) Integrazione del pattern

Per vedere come abbiamo utilizzato Guice nel nostro progetto, basta prendere una classe che ha un costruttore, come la classe `AccountImpl`.

```
@Inject
AccountImpl(Connection conn) {
    this.conn = conn;
}
```

Notiamo che il costruttore della classe dichiara delle dipendenze tra la classe `AccountImpl` e la classe `Connection`.
Il costruttore è marchato con l'annotazione @Inject per denotare che Guice sarà responasabile di fornire un'istanza di `Connection` quando si vorrà creare un'istanza di `AccountImp`.

La configurazione di del modulo Guice è fatta su un'altra file: `Init.java` che si occupa di soddisfare le dipendenze necesarie per la classe connection con il costrutto Injector.

```
Injector injector = Guice.createInjector(
    new DatabaseModule(),
    new WebserverModule(),
);
```

Nel momento in cui si volesse ottenere un'stanza di `AccountImpl` basterà chiamare il metodo getInstance cono l'oggetto injector creato precedentemente:

```
Injector injector = Guice.createInjector(/* ... */);
AccountImpl accountImpl = injector.getInstance(AccountImpl.class);
```

Con l'injector creato utilizzando Guice, è possibile ottenere un'istanza di `AccountImpl`. Guice si occuperà di soddisfare le dipendenze necessarie, come la `Connection`, iniettandole automaticamente nel costruttore di `AccountImpl`.

In generale, l'utilizzo di Guice semplifica la gestione delle dipendenze nel codice, separando la creazione delle istanze delle classi e la gestione delle dipendenze in un framework esterno, e viene integrato nel nostro prodotto per la maggioranza delle classi dotate di costruttore con delle dipendenze. Ciò rende il codice più modulare, facilitando la manutenzione e il testing.

4.3.2) Adapter

4.3.2.1) Motivazioni e studio del design pattern

Dato che abbiamo scelto di adottare un'architettura esagonale, diventa essenziale integrare il pattern adapter nel nostro sistema. Questo pattern svolge un ruolo fondamentale come interfaccia tra l'architettura esagonale e le componenti esterne, facilitando una collaborazione armoniosa all'interno dell'intero sistema. Grazie all'adapter siamo in grado di mantenere coerenza e indipendenza tra i vari moduli dell'architettura, garantendo al contempo un'integrazione senza problemi delle componenti esterne. Inoltre, questo approccio ci consente di mettere al centro del progetto la business logic, concentrandoci sulle funzionalità principali senza essere intralciati dalle complessità delle componenti esterne. Con adapter infatti possiamo trasformare le interfacce delle componenti esterne per adattarle alle nostre esigenze specifiche all'interno dell'architettura esagonale in maniera estramente agevole.

4.3.2.2) Implementazione del design pattern

Per l'implementazione di questo pattern abbiamo scelto di sviluppare internamente il codice necessario, evitando l'utilizzo di librerie o framework esterni, in modo da avere il controllo completo sul comportamento dell'adapter e poterlo personalizzare in base alle esigenze del progetto.

4.3.2.2.1) Utilizzo

Per ogni componente esterna con cui interagire abbiamo definito una o più porte di accesso al sistema, che altro non sono che una serie di interfacce contenenti i metodi necessari per comunicare con l'architettura esagonale, permettendo dunque l'isolamento della business logic. Successivamente, abbiamo definito una classe che implementi l'interfaccia dell'adapter, incaricata di adattare l'interfaccia delle componenti esterne a quella dell'architettura. All'interno di questa classe viene implementata la logica necessaria per convertire gli oggetti tra le due interfacce. Infine, l'adapter viene utilizzato all'interno dell'architettura esagonale per gestire la comunicazione con le componenti esterne.

4.3.2.3) Integrazione del pattern

Per una migliore comprensione dell'integrazione del pattern adapter nel nostro prodotto, possiamo esaminare una qualsiasi delle componenti esterne all'esagono insieme alla sua porta associata. Prendiamo ad esempio la classe EmailRepository e la relativa porta di accesso al sistema EmailPort:

```
public interface EmailPort {
  Email get(String id);
  Map<String, Email> getOf(String accountid);
  void insert(Email email);
  void replace(Email email);
  void delete(String id);
}
```

Partendo dalla porta, si vede facilmente come essa sia una semplice interfaccia che espone i metodi necessari per svolgere le classiche operazioni CRUD nel database riguardanti le email. Essa fornisce quindi un'astrazione per consentire la gestione delle email nel database, la quale può essere utilizzata dalle classi di business senza che quest'ultime debbano preoccuparsi dei dettagli di implementazione sottostanti.

```
public class EmailRepository implements EmailPort {
  private final RethinkDB r = RethinkDB.r;
  private Connection conn;
  private Gson gson;

  @Inject
  EmailRepository(Connection conn, Gson gson) {
    this.conn = conn;
    this.gson = gson;
  }

  @Override
  public Email get(String id) {
    String res = r.table("email")
                  .get(id)
                  .toJson()
                  .run(conn)
                  .single()
                  .toString();
    return gson.fromJson(res, Email.class);
  }

  @Override
  public Map<String, Email> getOf(String accountid) {
    ...
  }
```

```
  @Override
  public void insert(Email email) {
   r.table("email")
      .insert(gson.toJson(email))
      .run(conn);
  }


  @Override
  public void replace(Email email) {
    ...
  }

  @Override
  public void delete(String id) {
    ...
  }
}
```

La classe EmailRepository fornisce poi un'implementazione concreta dei metodi definiti nell'interfaccia sopra menzionata. Attualmente utilizziamo RethinkDB come database per eseguire tali operazioni, ma questa scelta non ha alcun impatto sulle classi di business. Pertanto, nel caso in cui decidessimo di cambiare il sistema di persistenza in futuro, sarebbe sufficiente aggiornare questa classe. I metodi infatti operano tramite oggetti di business passati come argomenti, tipo Email, consentendo di mantenere l'indipendenza dalle logiche di persistenza del prodotto.

In conclusione, quando ci troviamo ad interagire con componenti esterne adottiamo sempre adapter di questo tipo, basandoci sugli oggetti di business in modo che rimangano al centro di ogni operazione. Le porte permettono di definire un'interfaccia chiara e stabile per l'interazione con il mondo esterno, consentendo così una facile sostituzione o aggiornamento delle implementazioni sottostanti. D'altra parte, le loro implementazioni forniscono il collegamento reale tra l'esagono e i servizi esterni, gestendo dunque la logica di adattamento.

4.3.3) Builder

4.3.3.1) Motivazioni e studio del design pattern

Builder è uno dei design pattern creazionali più noti e utilizzati. Questo pattern è progettato per semplificare la creazione di oggetti complessi con molti attributi, guidando il processo di costruzione attraverso una serie di passaggi definiti tramite una catena di metodi. La sua principale utilità risiede nel separare la logica di costruzione di un oggetto complesso dalla sua rappresentazione, permettendo così di creare diverse rappresentazioni dello stesso oggetto mediante lo stesso processo di costruzione. La scelta di adottare il pattern builder da parte dello sviluppatore della libreria da noi utilizzata è quindi più che comprensibile, in quanto migliora l'organizzazione del processo di creazione degli oggetti e contribuisce a rendere il codice più leggibile, modulare e manutenibile.

4.3.3.2) Implementazione del design pattern

Il creatore della libreria ha saggiamente scelto di non implementare direttamente il pattern, bensì ha preferito affidarsi a Lombok, una libreria Java che offre una serie di annotazioni e strumenti per semplificare lo sviluppo riducendo la quantità di codice boilerplate che è necessario scrivere. Lombok semplifica l'implementazione del pattern builder introducendo l'annotazione @Builder, la quale, se applicata a una classe, farà si che, quando si compila il codice, venga generato automaticamente un builder interno per quella classe da parte della libreria. Questo builder consente di creare un'istanza della classe in questione in modo fluente, impostando i valori dei campi desiderati uno per uno.

4.3.3.2.1) Utilizzo

Vediamo quindi come lo sviluppatore della libreria utilizza Lombok per implementare il design pattern builder, prendendo come esempio la classe SessionResource:

```
package rs.ltt.jmap.common;

import java.util.Collection;
import java.util.Map;
import lombok.*;
import rs.ltt.jmap.common.entity.Account;
import rs.ltt.jmap.common.entity.AccountCapability;
import rs.ltt.jmap.common.entity.Capability;

@Builder
@Getter
@ToString
public class SessionResource {

    private String username;
    private String apiUrl;
    private String downloadUrl;
    private String uploadUrl;
    private String eventSourceUrl;
    @Singular private Map<String, Account> accounts;

    ...

    public static class SessionResourceBuilder {
        public SessionResourceBuilder capabilities(
                Map<Class<? extends Capability>, Capability> capabilities) {
            for (Map.Entry<Class<? extends Capability>, Capability> entry :
                    capabilities.entrySet()) {
                final Class<? extends Capability> key = entry.getKey();
                final Capability value = entry.getValue();
                if (key != value.getClass()) {
                    throw new IllegalArgumentException(
                            String.format(
                                    "key %s does not match value type %s", key,
value.getClass()));
                }
            }
            this.capabilities = capabilities;
            return this;
        }
    }
}
```

L'esempio illustra l'ampio utilizzo di Lombok nel codice. La classe SessionResource è marcata con l'annotazione @Builder, che genera automaticamente un builder per la classe stessa. Inoltre, in questo caso particolare, viene anche definito il metodo capabilities() all'interno della classe SessionResourceBuilder, il quale serve a impostare le capacità della sessione. In altri casi, invece, viene utilizzata soltanto l'annotazione @Builder.
Volendo, esaminando ulteriormente il codice della libreria, è possibile trovare altri esempi di utilizzo, poiché Lombok viene impiegato per la definizione di un builder per praticamente ogni oggetto.

4.3.3.3) Integrazione del pattern

Per andare a vedere come abbiamo integrato questo pattern nel nostro prodotto si può spaziare nelle varie classi che compongono la business logic di quest'ultimo, infatti quasi tutte andranno a costruire oggetti core della libreria e per farlo si affideranno alla catena di metodi dei builder relativi.
Prendiamo come esempio la creazione della risorsa JMAP Session all'interno della classe SessionLogic:

```
final SessionResource sessionResource =
        SessionResource.builder()
          .apiUrl("http://localhost:8000/api/jmap")
          .uploadUrl("http://localhost:8000/api/upload")
          .downloadUrl("http://localhost:8000/api/download" + "?blobid={blobId}")
          .state(accountPort.getState(username))
          .username(username)
          .eventSourceUrl("")
          .account(
              accountid,
              Account.builder()
                .accountCapabilities(
                  ImmutableMap.of(
                      MailAccountCapability.class,
                      MailAccountCapability.builder()
                          .maxSizeAttachmentsPerEmail(50 * 1024 * 1024L) // 50MiB
                          .build()))
                .isPersonal(true)
                .isReadOnly(false)
                .build())
          .capabilities(capabilityBuilder.build())
          .primaryAccounts(ImmutableMap.of(MailAccountCapability.class, accountid))
          .build();
```

In questo esempio, possiamo osservare come l'oggetto SessionResource, definito dalla libreria, venga costruito in modo chiaro e conciso attraverso una catena di metodi. Questa catena di chiamate imposta i valori desiderati come i diversi URL, lo stato della sessione e lo username dell'utente. È interessante notare che alcuni parametri vengono costruiti a loro volta utilizzando dei builder dedicati, come ad esempio l'oggetto Account, all'interno del quale un'altra catena di metodi ancora crea l'oggetto MailAccountCapability della libreria.
Si noti come l'ampio utilizzo del pattern builder da parte dello sviluppatore della libreria porti a un codice più leggibile, flessibile e manutenibile. Questo riduce notevolmente la complessità associata alla creazione di oggetti con un gran numero di parametri, come dimostrato nell'esempio. Inoltre, l'uso del pattern builder ci consente di evitare la configurazione dei parametri opzionali della sessione quando non sono necessari, semplificando ulteriormente il processo di creazione degli oggetti.

In conclusione, l'integrazione del pattern builder nel nostro prodotto è una pratica comune ogni volta che è necessario costruire un oggetto della libreria con un builder associato. Questo migliora l'efficienza dello sviluppo e la qualità del nostro software, garantendo un codice più pulito e manutenibile.

4.4) Diagramma delle classi

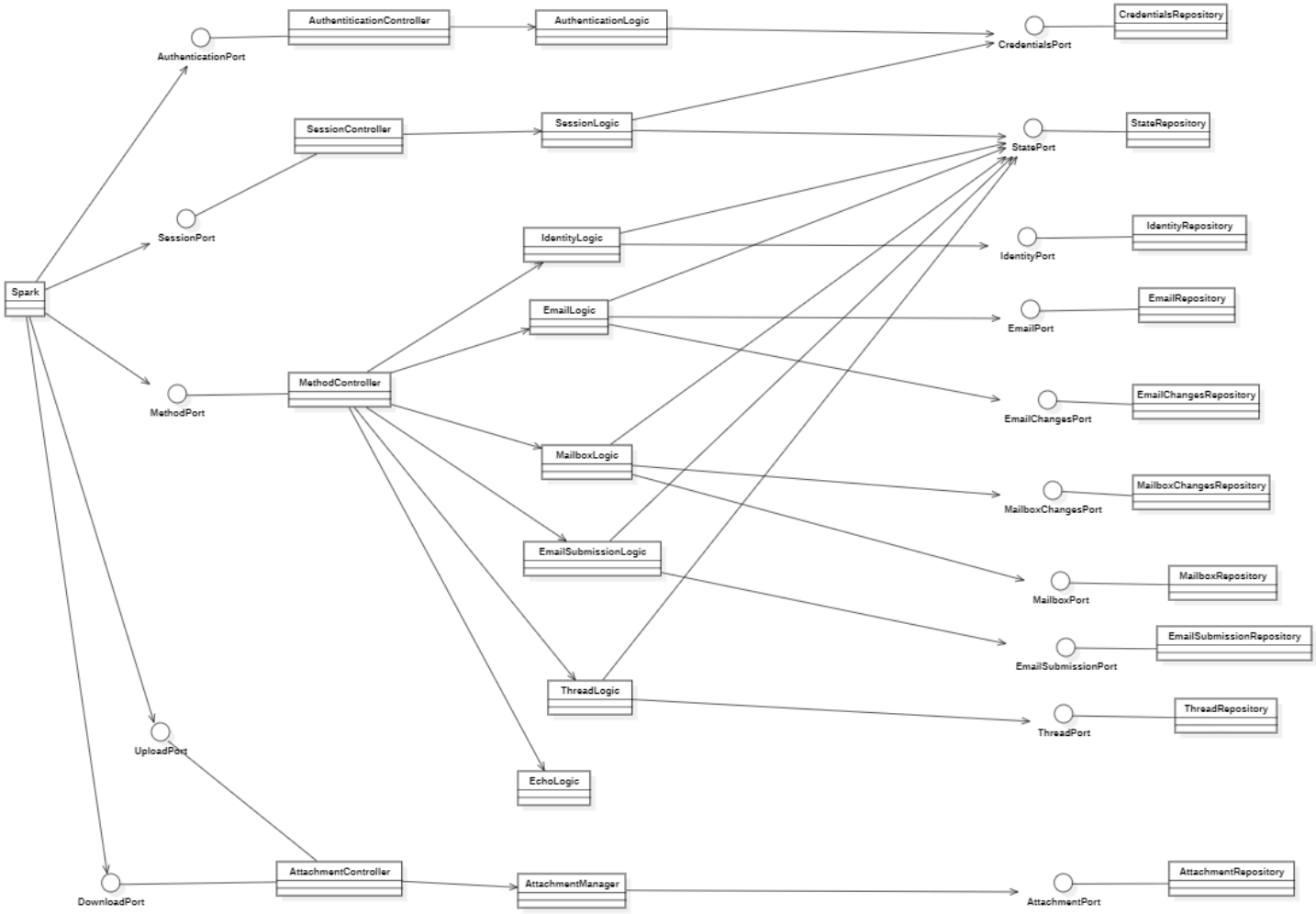

Figura 3: Diagramma delle classi

4.4.1) Ingresso delle richieste nell'applicazione

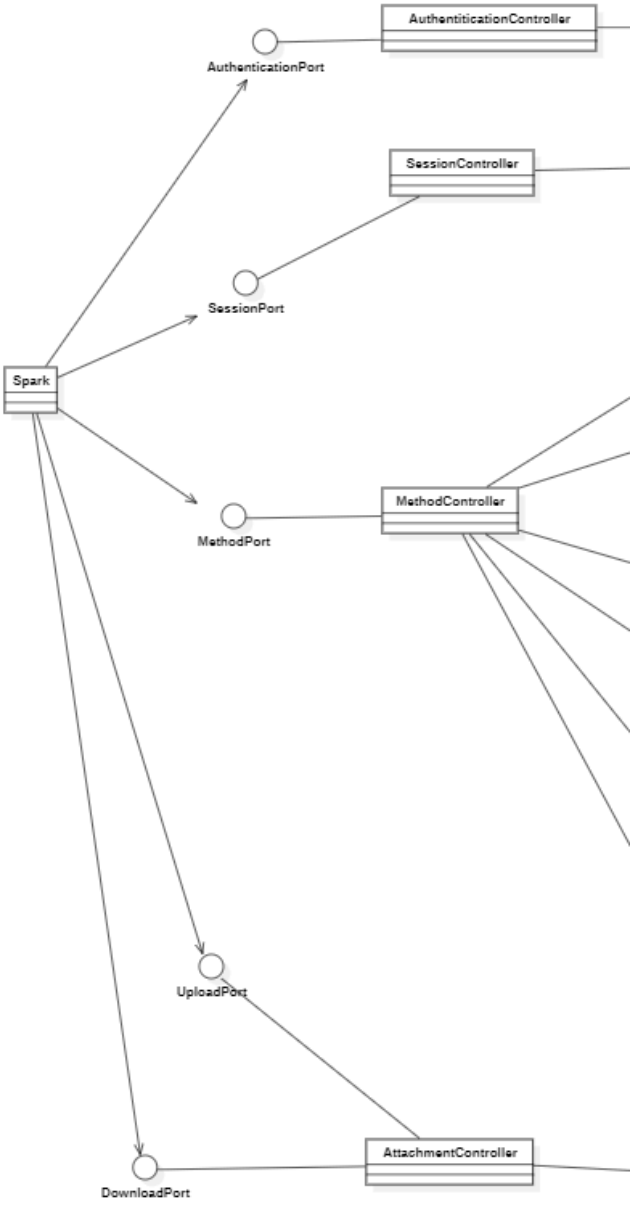

Figura 4: Modellazione delle componenti che gestiscono l'ingresso delle richieste nell'applicazione

Il pezzo di diagramma riportato illustra le componenti fondamentali per gestire l'ingresso delle richieste provenienti dai client all'interno del sistema. Vi si trovano le seguenti classi:

- **Spark**: gestore principale dell'ingresso delle richieste in arrivo (input), il cui scopo è quello di definire molteplici rotte utilizzando il framework Spark. Include al suo interno la gestione iniziale delle operazioni come l'autenticazione, la gestione delle sessioni, la gestione dei metodi di JMAP e la manipolazione degli allegati attraverso upload e download.
  - *Proprietà*:
    - authentication: AuthenticationPort - Un'istanza di AuthenticationPort utilizzata per gestire l'autenticazione degli utenti;
    - session: SessionPort - Un'istanza di SessionPort utilizzata per gestire le sessioni degli utenti;
    - method: MethodPort - Un'istanza di MethodPort utilizzata per gestire i metodi di JMAP;
    - upload: UploadPort - Un'istanza di UploadPort utilizzata per gestire le operazioni di caricamento degli allegati;

- download: DownloadPort - Un'istanza di DownloadPort utilizzata per gestire le operazioni di scaricamento degli allegati.

▸ *Operazioni*:

- `authenticate(q: Request, a: Response): String` - Verifica dell'autenticazione dell'utente;
- `up(q: Request, a: Response): String` - Verifica della corretta esecuzione del server;
- `wellKnown(q: Request, a: Response): String` - Effettua un reindirizzamento alla rotta "/api/jmap";
- `getJmap(q: Request, a: Response): String` - Gestisce le richieste GET per la rotta "/api/jmap" e restituisce i dati di sessione in formato JSON;
- `postJmap(q: Request, a: Response): String` - Gestisce le richieste POST per la rotta "/api/jmap" e restituisce i dati elaborati in base al corpo della richiesta;
- `download(q: Request, a: Response): String` - Gestisce le richieste GET per la rotta "/api/download" riguardanti gli allegati e restituisce il contenuto richiesto;
- `upload(q: Request, a: Response): String` - Gestisce le richieste POST per la rotta "/api/download" riguardanti gli allegati e li carica nel server;
+ `start(): void` - Configura e avvia il server Spark, impostando la porta e definendo i filtri e le rotte con i rispettivi handler di richieste HTTP.

- **AuthenticationPort**: porta in ingresso che funge da punto di accesso per l'interazione tra il dominio dell'applicazione (core) e il mondo esterno. Essa definisce un insieme di operazioni che rappresentano le azioni che l'applicazione può eseguire in risposta alle richieste esterne provenienti dai client riguardanti l'autenticazione. La sua implementazione è responsabilità di classi concrete che forniranno la logica specifica per gestire tali operazioni in modo efficace all'interno dell'applicazione, mantenendo così la separazione tra i diversi tipi di logiche.

  ▸ *Operazioni*:

  # `authenticate(auth: String): boolean` - Restituisce un valore booleano che indica se l'autenticazione è riuscita o meno. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia, in modo che possano definire la logica specifica per verificare l'autenticità delle credenziali fornite.

- **SessionPort**: porta in ingresso che funge da punto di accesso per l'interazione tra il dominio dell'applicazione (core) e il mondo esterno. Essa definisce un insieme di operazioni che rappresentano le azioni che l'applicazione può eseguire in risposta alle richieste esterne provenienti dai client riguardanti la sessione. La sua implementazione è responsabilità di classi concrete che forniranno la logica specifica per gestire tali operazioni in modo efficace all'interno dell'applicazione, mantenendo così la separazione tra i diversi tipi di logiche.

  ▸ *Operazioni*:

  # `get(auth: String): String` - Restituisce una stringa che rappresenta le informazioni di sessione. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia, in modo che possano definire la logica specifica per ottenere le informazioni di sessione in base alle credenziali fornite.

- **MethodPort**: porta in ingresso che funge da punto di accesso per l'interazione tra il dominio dell'applicazione (core) e il mondo esterno. Essa definisce un insieme di operazioni che rappresentano le azioni che l'applicazione può eseguire in risposta alle richieste esterne provenienti dai client riguardanti i vari metodi del protocollo JMAP. La sua implementazione è responsabilità di classi concrete che forniranno la logica specifica per gestire tali operazioni in modo efficace all'interno dell'applicazione, mantenendo così la separazione tra i diversi tipi di logiche.

  ▸ *Operazioni*:

  # `dispatch(in: String): String` - Prende una stringa come parametro, la quale rappresenta una richiesta in formato JSON, la elabora e restituisce una stringa che rappresenta la risposta alla richiesta. L'implementazione concreta di questo metodo è lasciata alle classi che implementano

questa interfaccia, in modo che possano definire la logica specifica per l'elaborazione delle richieste.

- **UploadPort**: porta in ingresso che funge da punto di accesso per l'interazione tra il dominio dell'applicazione (core) e il mondo esterno. Essa definisce un insieme di operazioni che rappresentano le azioni che l'applicazione può eseguire in risposta alle richieste esterne provenienti dai client riguardanti l'upload di allegati. La sua implementazione è responsabilità di classi concrete che forniranno la logica specifica per gestire tali operazioni in modo efficace all'interno dell'applicazione, mantenendo così la separazione tra i diversi tipi di logiche.
  - *Operazioni*:
    `# push(data: byte[]): String` - Prende un array di byte come parametro, ovvero i dati che devono essere caricati relativi agli allegati. La richiesta di upload verrà poi elaborata e verrà restituita una stringa rappresentante un identificatore univoco assegnato al dato caricato. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia, in modo che possano definire la logica specifica per il caricamento dei dati degli allegati.
- **DownloadPort**: porta in ingresso che funge da punto di accesso per l'interazione tra il dominio dell'applicazione (core) e il mondo esterno. Essa definisce un insieme di operazioni che rappresentano le azioni che l'applicazione può eseguire in risposta alle richieste esterne provenienti dai client riguardanti il download di allegati. La sua implementazione è responsabilità di classi concrete che forniranno la logica specifica per gestire tali operazioni in modo efficace all'interno dell'applicazione, mantenendo così la separazione tra i diversi tipi di logiche.
  - *Operazioni*:
    `# pull(id: String): byte[]` - Prende una stringa come parametro corrispondente ad un identificatore univoco assegnato ai dati di un allegato precedentemente caricato e che ora si vuole scaricare. La richiesta di download verrà poi elaborata cercando i dati dell'allegato associati a quell'identificatore, i quali verranno restituiti come un array di byte. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia, in modo che possano definire la logica specifica per il recupero dei dati in base all'identificatore fornito.
- **AuthenticationController**: implementazione dell'interfaccia AuthenticationPort. Essenzialmente, funge da intermediario tra le richieste provenienti dall'esterno dell'applicazione rigurdanti l'autenticazione e la logica di business sottostante, gestendo quindi la parte di application logic relativa. Al suo interno si esegue dunque una prima validazione delle richieste di questo tipo, in modo da evitare di arrivare alla business logic nel caso in cui il formato della richiesta non fosse conforme allo standard.
  - *Proprietà*:
    - authenticationLogic: AuthenticationLogic - Un'istanza di AuthenticationLogic utilizzata per gestire la logica di business riguardante l'autenticazione degli utenti.
  - *Operazioni*:
    `+ authenticate(auth: String): boolean` - Implementa il metodo authenticate dell'interfaccia AuthenticationPort. Esegue una prima validazione della richiesta prima di passarla alla classe di business relativa.
- **SessionController**: implementazione dell'interfaccia SessionPort. Essenzialmente, funge da intermediario tra le richieste provenienti dall'esterno dell'applicazione rigurdanti la sessione e la logica di business sottostante, gestendo quindi la parte di application logic relativa. Al suo interno si esegue dunque una prima validazione delle richieste di questo tipo, in modo da evitare di arrivare alla business logic nel caso in cui il formato della richiesta non fosse conforme allo standard.
  - *Proprietà*:
    - gson: Gson - Un'istanza di Gson utilizzata per serializzare gli oggetti in formato JSON;
    - sessionLogic: SessionLogic - Un'istanza di SessionLogic utilizzata per gestire la logica di business riguardante la sessione.

- *Operazioni*:
  + `get(auth: String): String` - Implementa il metodo get dell'interfaccia SessionPort. Prende in input una stringa che contiene le credenziali di autenticazione, estrae il token di autenticazione e decodifica quest'ultimo utilizzando Base64 per ottenere il nome utente e la password, validando la parte di application logic. Successivamente passa il nome utente estratto all'istanza di SessionLogic posseduta, la quale si occuperà di gestire la business logic, e serializza l'oggetto restituito da quest'ultima in formato JSON utilizzando l'istanza di Gson, restituendo infine il JSON risultante dalle operazioni eseguite.

- **MethodController**: implementazione dell'interfaccia MethodPort. Essenzialmente, funge da intermediario tra le richieste provenienti dall'esterno dell'applicazione rigurdanti i vari metodi del protocollo JMAP e la logica di business sottostante, gestendo quindi la parte di application logic relativa. Al suo interno si esegue dunque una prima validazione delle richieste di questo tipo, in modo da evitare di arrivare alla business logic nel caso in cui il formato della richiesta non fosse conforme allo standard. Riceve le richieste in arrivo e in base al tipo di richiesta e ai dati associati, determina come instradarla all'interno dell'applicazione.
  - *Proprietà*:
    - gson: Gson - Un'istanza di Gson utilizzata per serializzare gli oggetti in formato JSON;
    - echo: EchoLogic - Un'istanza di EchoLogic utilizzata per gestire la logica di business riguardante le richieste di eco;
    - email: EmailLogic - Un'istanza di EmailLogic utilizzata per gestire la logica di business riguardante le operazioni relative alle email;
    - submission: EmailSubmissionLogic - Un'istanza di EmailSubmissionLogic utilizzata per gestire la logica di business riguardante le operazioni relative all'invio di email;
    - identity: IdentityLogic - Un'istanza di IdentityLogic utilizzata per gestire la logica di business riguardante le operazioni relative all'identità;
    - mailbox: MailboxLogic - Un'istanza di MailboxLogic utilizzata per gestire la logica di business riguardante le operazioni relative alle caselle di posta;
    - thread: ThreadLogic - Un'istanza di ThreadLogic utilizzata per gestire la logica di business riguardante le operazioni relative ai thread di email.
  - *Operazioni*:
    # `dispatch(in: String): String` - Implementa il metodo dispatch dell'interfaccia MethodPort. Prende in input una stringa contenente una richiesta JMAP, estrae le chiamate di metodo dalla richiesta, esegue una prima validazione di quest'ultime e determina come instradarle all'interno dell'applicazione utilizzando il metodo privato pick. Infine aggrega le risposte ottenute dalle classi di business in una risposta JMAP complessiva e restituisce la rappresentazione JSON di quest'ultima;
    - `pick(methodCall: MethodCall, prevResponses: ListMultimap\<String, Response.Invocation>): MethodResponse[]` - Prende in input una MethodCall e una mappa delle risposte precedenti in modo da determinare quale metodo della logica di business corrispondente chiamare per poter elaborare la richiesta ed infine restituisce la risposta corrispondente. Se il tipo di MethodCall è sconosciuto, restituisce una risposta di errore.

- **AttachmentController**: implementazione delle interfacce UploadPort e DownloadPort. Essenzialmente, funge da intermediario tra le richieste provenienti dall'esterno dell'applicazione rigurdanti l'upload o il download degli allegati e la logica di business sottostante, gestendo quindi la parte di application logic relativa. Al suo interno si esegue dunque una prima validazione delle richieste di questo tipo, in modo da evitare di arrivare alla business logic nel caso in cui il formato della richiesta non fosse conforme allo standard.
  - *Proprietà*:
    - gson: Gson - Un'istanza di Gson utilizzata per serializzare gli oggetti in formato JSON;

- attachmentLogic: AttachmentLogic - Un'istanza di AttachmentLogic utilizzata per gestire la logica di business riguardante la gestione degli allegati nell'applicazione.

- *Operazioni*:
  + `pull(id: String): byte[]` - Implementa il metodo pull dell'interfaccia DownloadPort. Prende in input un identificatore e utilizza l'istanza di AttachmentLogic per scaricare l'allegato corrispondente. Restituisce quindi i dati dell'allegato sotto forma di array di byte;
  + `push(data: byte[]): String` - Implementa il metodo push dell'interfaccia UploadPort. Prende in input un array di byte rappresentante i dati dell'allegato da caricare e utilizza l'istanza di AttachmentLogic per eseguire il caricamento dell'allegato e ottenere una risposta, corrispondente ad un identificatore univoco assegnato al dato. Questa risposta viene quindi serializzata in formato JSON utilizzando l'istanza di Gson e restituita come una stringa JSON.

4.4.2) Gestione delle richieste

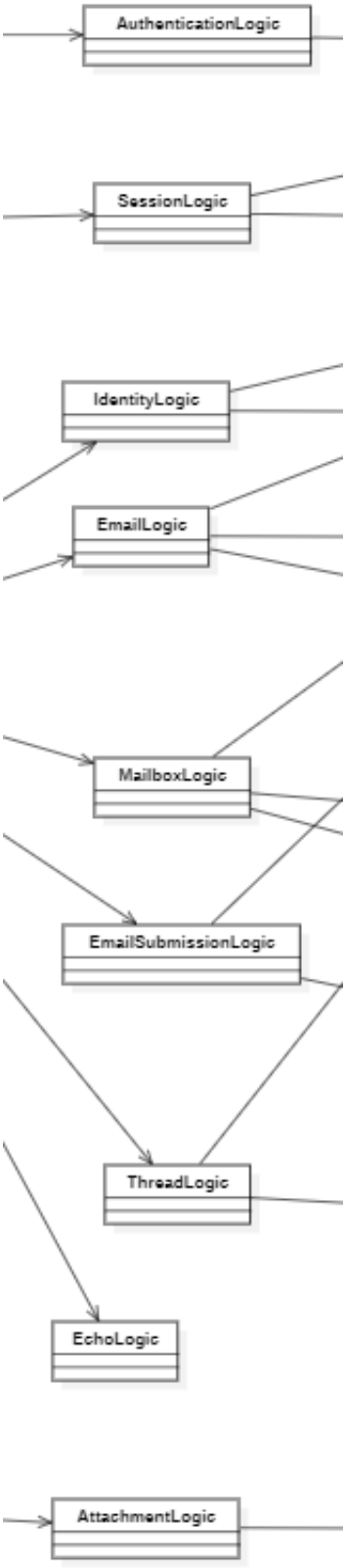

Figura 5: Modellazione delle componenti che gestiscono le rischieste all'interno dell'applicazione

Le componenti riportate nel frammento di diagramma soprastante sono quelle necessarie per la gestione delle richieste, implementando dunque la business logic del prodotto. Vi si trovano le seguenti classi:

- **AuthenticationLogic**: componente responsabile della gestione dell'autenticazione all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:
    - account: AccountPort - Un'istanza di AccountPort utilizzata per operare con gli account degli utenti all'interno del database.
  - *Operazioni*:
    `# authenticate(username: String, password: String): boolean` - Riceve in input un nome utente e una password e verifica se le credenziali fornite corrispondono a quelle memorizzate nel sistema utilizzando l'istanza di AccountPort.
- **SessionLogic**: componente responsabile della gestione delle sessioni all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:
    - account: AccountPort - Un'istanza di AccountPort utilizzata per operare con gli account degli utenti all'interno del database;
    - statePort: StatePort - Un'istanza di StatePort utilizzata per svolgere operazioni riguardanti lo stato dei client all'interno del database.
  - *Operazioni*:
    `# get(username: String): SessionResource` - Riceve in input un nome utente e crea una risorsa JMAP Session associata a quell'utente. La risorsa di sessione contiene informazioni quali URL dell'API, URL di upload e download, stato dell'account, elenco delle capacità supportate e altre informazioni relative alla sessione. Restituisce infine la risorsa di sessione creata.
- **IdentityLogic**: componente responsabile della gestione delle identità all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:
    - identityPort: IdentityPort - Un'istanza di IdentityPort utilizzata per l'accesso alle informazioni sulle identità, nonché per svolgere operazioni su quest'ultime all'interno del database;
  - *Operazioni*:
    `# get(methodCall: GetIdentityMethodCall, previousResponses: ListMultimap\<String, Response.Invocation>): MethodResponse[]` - Riceve in input una chiamata di metodo GetIdentityMethodCall e una mappa di risposte precedenti. Restituisce un array di MethodResponse che contiene le informazioni richieste sull'identità dell'utente. Quest'ultime includono l'id dell'account e altre informazioni aggiuntive che possono essere recuperate attraverso l'istanza di IdentityPort.
- **EmailLogic**: componente responsabile della gestione delle email all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo, tra cui anche l'implementazione di un sistema di sincronizzazione che permetta ad un client di mantenersi aggiornato con gli ultimi aggiornamenti della casella di posta visualizzata per quanto riguarda le email.
  - *Proprietà*:
    - mailboxPort: MailboxPort - Un'istanza di MailboxPort utilizzata per svolgere operazioni sulle caselle di posta all'interno del database;
    - emailPort: EmailPort - Un'istanza di EmailPort utilizzata per svolgere operazioni sulle email all'interno del database;
    - statePort: StatePort - Un'istanza di StatePort utilizzata per svolgere operazioni riguardanti lo stato dei client all'interno del database;
    - mailboxUpdatePort: MailboxUpdatePort - Un'istanza di MailboxUpdatePort utilizzata per svolgere operazioni sui cambiamenti avvenuti alle caselle di posta all'interno del database.

- *Operazioni*:
  # `get(methodCall: GetEmailMethodCall, previousResponses: ListMultimap<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta GetEmailMethodCall e restituisce una lista di risposte di tipo MethodResponse. Recupera un insieme di email in base a una specifica richiesta GetEmailMethodCall, la quale contiene gli identificativi delle email da reperire. Riceve informazioni sulle chiamate precedenti per ragioni di efficienza, in modo che la risposta ad una richiesta precedente possa essere utilizzata in input per la richiesta corrente. Se vengono specificate proprietà specifiche da recuperare, vengono recuperate solo quelle proprietà per le email restituite;
  # `query(methodCall: QueryEmailMethodCall, previousResponses: ListMultimap<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta QueryEmailMethodCall e restituisce una lista di risposte di tipo MethodResponse. Esegue una query sulle email corrispondenti ai criteri specificati nella richiesta. La query può essere filtrata in base a diversi criteri come la casella di posta, l'oggetto, i destinatari, e così via. Riceve informazioni sulle chiamate precedenti per ragioni di efficienza, in modo che la risposta ad una richiesta precedente possa essere utilizzata in input per la richiesta corrente;
  # `set(methodCall: SetEmailMethodCall, previousResponses: ListMultimap<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta SetEmailMethodCall e restituisce una lista di risposte di tipo MethodResponse. La richiesta può contenere un elenco di email da aggiornare, creare o eliminare. Utilizza l'EmailPort per eseguire le modifiche richieste, ad esempio aggiornando le proprietà delle email esistenti o creandone di nuove. Riceve un elenco di risposte di invocazione precedenti per ragioni di efficienza, in modo che la risposta ad una richiesta precedente possa essere utilizzata in input per la richiesta corrente, e restituisce una lista di risposte che indicano lo stato delle operazioni eseguite;
  # `changes(methodCall: ChangesEmailMethodCall, previousResponses: ListMultimap<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta ChangesEmailMethodCall e restituisce una lista di risposte di tipo MethodResponse. Consente di ottenere i cambiamenti avvenuti nelle email da un certo punto temporale in poi, in base all'ultimo stato noto contenuto nella richiesta fornita;
  - `patchEmail(String, Map<String, Object>, ListMultimap<String, Response.Invocation>): Email` - Riceve le modifiche da applicare sotto forma di una mappa di chiavi e valori, e restituisce l'email modificata;
  - `processCreateEmail(Map<String, Email>, SetEmailMethodResponse.SetEmailMethodResponseBuilder, ListMultimap<String, Response.Invocation>, String): void` - Gestisce la creazione di nuove email. Prende in input una mappa di email da creare, le elabora e le inserisce nel database attraverso l'EmailPort associando le email all'account specificato. Se è presente un identificativo di creazione, viene utilizzato quest'ultimo per associare la risposta di creazione all'identificativo specificato;
  - `getAccumulatedUpdateSince(String, String): Update` - Restituisce gli aggiornamenti accumulati per un account riguardanti le email a partire dalla versione specificata;
  - `injectId(Attachment): EmailBodyPart` - Inietta un nuovo identificativo in un allegato email. Riceve un oggetto Attachment e restituisce un oggetto EmailBodyPart con un identificativo generato casualmente, utilizzato per identificare l'allegato;
  - `applyFilter(Filter<Email>, Stream<Email>): Stream<Email>` - Applica un filtro agli oggetti Email all'interno dello stream fornito;
  - `distinctByKey(Function<? super T, ?>): Predicate<T>` - Restituisce un predicato che filtra gli oggetti in base a una chiave estratta da ognuno di essi, garantendo che siano distinti in base alla chiave.

- **MailboxLogic**: componente responsabile della gestione delle caselle di posta all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo, tra cui anche l'implementazione di un sistema di sincronizzazione che permetta ad un client di mantenersi aggiornato con gli ultimi aggiornamenti della casella di posta visualizzata per quanto riguarda le cartelle.
  - *Proprietà*:
    - emailPort: EmailPort - Un'istanza di EmailPort utilizzata per svolgere operazioni sulle email all'interno del database;
    - mailboxPort: MailboxPort - Un'istanza di MailboxPort utilizzata per svolgere operazioni sulle caselle di posta all'interno del database;
    - statePort: StatePort - Un'istanza di StatePort utilizzata per svolgere operazioni riguardanti lo stato dei client all'interno del database;
    - mailboxUpdatePort: MailboxUpdatePort - Un'istanza di MailboxUpdatePort utilizzata per svolgere operazioni sui cambiamenti avvenuti alle cartelle all'interno del database.
  - *Operazioni*:
    `# get(GetMailboxMethodCall, ListMultimap<String, Response.Invocation>): MethodResponse[]` - Ottiene le informazioni delle caselle di posta specificate nella richiesta. Riceve una chiamata GetMailboxMethodCall, che specifica le caselle di posta di interesse, e restituisce una serie di risposte che includono le informazioni richieste e lo stato aggiornato;
    `# set(SetMailboxMethodCall, ListMultimap<String, Response.Invocation>): MethodResponse[]` - Gestisce le richieste di impostazione delle caselle di posta, inclusa la creazione e la modifica. Riceve una chiamata SetMailboxMethodCall, che contiene le informazioni sulla creazione o la modifica delle cartelle, e restituisce una serie di risposte che includono lo stato aggiornato delle caselle di posta in risposta alle richieste;
    `# changes(ChangesMailboxMethodCall, ListMultimap<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta ChangesMailboxMethodCall e restituisce una lista di risposte di tipo MethodResponse. Consente di ottenere i cambiamenti avvenuti nelle cartelle da un certo punto temporale in poi, in base all'ultimo stato noto contenuto nella richiesta fornita;
    - `getAccumulatedUpdateSince(String, String): Update` - Restituisce gli aggiornamenti accumulati per un account riguardanti le cartelle a partire dalla versione specificata;
    - `toMailbox(MailboxInfo, String): Mailbox` - Converte un oggetto MailboxInfo nel formato richiesto per la risposta del metodo get, includendo le informazioni aggiuntive sul numero di email e thread nelle caselle di posta;
    - `processCreateMailbox(Map<String, Mailbox>, SetMailboxMethodResponse.SetMailboxMethodResponseBuilder, String): void` - Gestisce la creazione di nuove caselle di posta, controllando se esistono caselle di posta con lo stesso nome e aggiornando di conseguenza la risposta;
    - `processUpdateMailbox(Map<String, Map<String, Object>>, SetMailboxMethodResponse.SetMailboxMethodResponseBuilder, ListMultimap<String, Response.Invocation>, String): void` - Gestisce la modifica delle caselle di posta esistenti e aggiorna di conseguenza la risposta, controllando le modifiche e applicandole ai valori esistenti delle caselle di posta;
    - `patchMailbox(String, Map<String, Object>, ListMultimap<String, Response.Invocation>, String): MailboxInfo` - Applica le modifiche specificate a una casella di posta esistente, come la modifica del ruolo della casella di posta, e restituisce le informazioni aggiornate della casella di posta.
- **EmailSubmissionLogic**: componente responsabile della gestione dell'invio di un'email per la consegna a uno o più destinatari all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:

- *Operazioni*:

- **ThreadLogic**: componente responsabile della gestione dei thread all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:
    - threadPort: ThreadPort - Un'istanza di ThreadPort utilizzata per svolgere operazioni sui thread all'interno del database;
    - threadUpdatePort: ThreadUpdatePort - Un'istanza di ThreadUpdatePort utilizzata per l'accesso alle informazioni sugli aggiornamenti dei thread e per svolgere operazioni su di essi all'interno del database;
    - statePort: StatePort - Un'istanza di StatePort utilizzata per svolgere operazioni riguardanti lo stato dei client all'interno del database;
    - emailPort: EmailPort - Un'istanza di EmailPort utilizzata per svolgere operazioni sulle email all'interno del database.
  - *Operazioni*:
    `# changes(methodCall: ChangesThreadMethodCall, previousResponses: ListMultimap\<String, Response.Invocation>): MethodResponse[]` - Riceve una richiesta ChangesThreadMethodCall e restituisce una lista di risposte di tipo MethodResponse. Consente di ottenere i cambiamenti avvenuti nei thread da un certo punto temporale in poi, in base all'ultimo stato noto contenuto nella richiesta fornita;
    `# get(methodCall: GetThreadMethodCall, previousResponses: ListMultimap\<String, Response.Invocation>): MethodResponse[]` - Ottiene le informazioni sui thread delle email specificati nella richiesta. Riceve una chiamata GetThreadMethodCall, che specifica i thread di interesse, e restituisce una serie di risposte che includono le informazioni richieste e lo stato aggiornato;
    - `getAccumulatedUpdateSince(oldVersion: String, accountid: String): Update` - Restituisce gli aggiornamenti accumulati per un account riguardanti i thread a partire dalla versione specificata.
- **EchoLogic**: componente responsabile della gestione degli echo per testare la connettività all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Operazioni*:
    `# echo(methodCall: EchoMethodCall, previousResponses: ListMultimap\<String, Response.Invocation>): MethodResponse[]`- Gestisce la logica di esecuzione della chiamata EchoMethodCall. Restituisce un array contenente la risposta di echo.
- **AttachmentLogic**: componente responsabile della gestione degli allegati all'interno del sistema. Si occupa quindi di tutte le operazioni relative alla business logic di questo tipo.
  - *Proprietà*:
    - attachmentPort: AttachmentPort - Un'istanza di AttachmentPort utilizzata per svolgere operazioni sugli allegati all'interno del database.
  - *Operazioni*:
    `# upload(data: byte[]): Upload` - Carica i dati dell'allegato forniti utilizzando la porta attachmentPort e restituisce un oggetto Upload contenente l'identificativo del blob caricato;
    `# download(id: String): byte[]` - Scarica i dati dell'allegato con l'identificativo fornito utilizzando la porta attachmentPort e restituisce i dati come array di byte.

4.4.3) Interfacciamento al database

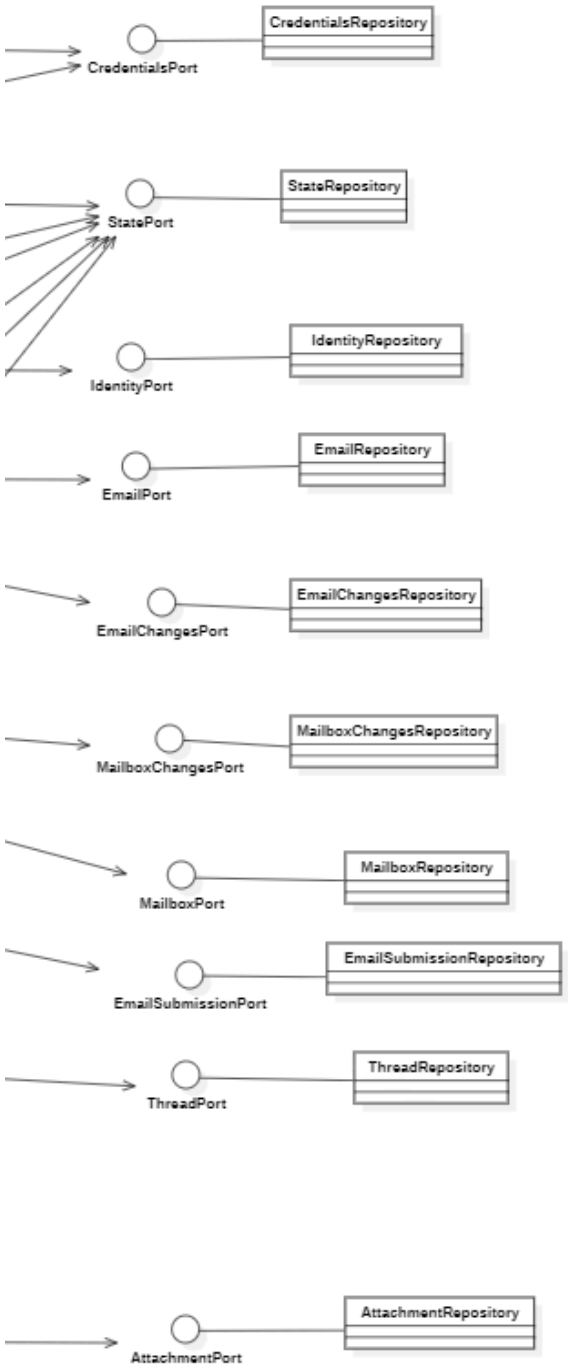

Figura 6: Modellazione delle componenti che gestiscono l'interfacciamento al database

Nell'ultima parte del diagramma delle classi si trovano le componenti dedicate alla gestione dell'interfacciamento con il database. Qui si incontrano, quindi, tutte le porte in uscita dall'esagono contenente la business logic e le classi concrete che svolgono operazioni specifiche su vari tipi di dati con il database. Nello specifico queste sono le seguenti:

- **AccountPort**: porta in uscita che definisce una serie di metodi per operare con gli account degli utenti all'interno del database.
  - *Operazioni*:
    `# getId(username: String): String` - Prende in input un parametro che rappresenta lo username dell'account di cui si desidera ottenere l'identificativo. Restituisce una stringa che rappresenta

l'identificativo associato all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
`getPassword(id: String): String` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desidera ottenere la password. Restituisce una stringa che rappresenta la password corrente associata all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.

- **StatePort**: porta in uscita che definisce una serie di metodi per svolgere operazioni riguardanti lo stato dei client all'interno del database.
  - *Operazioni*:
    # `getState(accountid: String): String` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desidera ottenere lo stato. Restituisce una stringa che rappresenta lo stato corrente associato all'account specificato. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia;
    # `incrementState(accountid: String): void` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desidera incrementare lo stato e lo aggiorna. L'implementazione concreta di questo metodo è lasciata alle classi che implementano questa interfaccia.
- **IdentityPort**: porta in uscita che definisce una serie di metodi per l'accesso alle informazioni sulle identità, nonché per svolgere operazioni su quest'ultime all'interno del database.
  - *Operazioni*:
    # `getOf(accountid: String): Identity[]` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desidera ottenere le informazioni sulle identità associate. Restituisce un array di oggetti Identity che rappresentano le identità associate all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **EmailPort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sulle email all'interno del database;
  - *Operazioni*:
    # `get(id: String): Email` - Prende in input un parametro che rappresenta l'identificativo univoco di un'email e restituisce l'oggetto Email corrispondente. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    # `getOf(accountid: String): Map<String, Email>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere le email associate e restituisce una mappa che associa gli identificatori univoci delle email ai rispettivi oggetti Email. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    # `insert(accountid: String, email: Email): void` - Prende in input due parametri, l'identificatore dell'account e un oggetto Email da inserire, e aggiunge l'email associata all'account specificato nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    # `delete(id: String): void` - Prende in input un parametro che rappresenta l'identificativo univoco di un'email e elimina l'email corrispondente dal database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **EmailUpdatePort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sui cambiamenti avvenuti alle email all'interno del database;
  - *Operazioni*:
    # `get(accountid: String, state: String): Update` - Prende in input due parametri, l'identificatore dell'account e lo stato corrente delle email, e restituisce l'oggetto Update relativo alle modifiche apportate alle email associate all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    # `getOf(accountid: String): Map<String, Update>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere gli aggiornamenti delle email e

restituisce una mappa che associa gli identificatori univoci delle email agli oggetti Update corrispondenti. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
`# insert(accountid: String, oldstate: String, update: Update): String` - Prende in input tre parametri, l'identificatore dell'account, lo stato precedente delle email e un oggetto Update rappresentante le modifiche apportate alle email, e restituisce una stringa che rappresenta un identificativo univoco dell'operazione di aggiornamento delle email inserita nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.

- **MailboxUpdatePort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sui cambiamenti avvenuti alle cartelle all'interno del database;
  - *Operazioni*:
    `# get(accountid: String, state: String): Update` - Prende in input due parametri, l'identificatore dell'account e lo stato corrente delle caselle di posta elettronica, e restituisce l'oggetto Update relativo alle modifiche apportate alle caselle di posta associate all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# getOf(accountid: String): Map<String, Update>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere gli aggiornamenti delle caselle di posta elettronica e restituisce una mappa che associa gli identificatori univoci delle caselle di posta agli oggetti Update corrispondenti. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# insert(accountid: String, oldstate: String, update: Update): String` - Prende in input tre parametri, l'identificatore dell'account, lo stato precedente delle caselle di posta elettronica e un oggetto Update rappresentante le modifiche apportate alle caselle di posta, e restituisce una stringa che rappresenta un identificativo univoco dell'operazione di aggiornamento delle caselle di posta inserita nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **MailboxPort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sulle caselle di posta all'interno del database;
  - *Operazioni*:
    `# get(id: String): MailboxInfo` - Prende in input un parametro che rappresenta l'identificativo univoco di una casella di posta elettronica e restituisce l'oggetto MailboxInfo corrispondente. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# getOf(accountid: String): Map<String, MailboxInfo>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere le caselle di posta elettronica associate e restituisce una mappa che associa gli identificatori univoci delle caselle di posta ai rispettivi oggetti MailboxInfo. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# insert(accountid: String, mailbox: MailboxInfo): String` - Prende in input due parametri, l'identificatore dell'account e un oggetto MailboxInfo da inserire, e restituisce una stringa che rappresenta l'identificativo univoco della casella di posta inserita nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# delete(id: String): void` - Prende in input un parametro che rappresenta l'identificativo univoco di una casella di posta elettronica e elimina la casella di posta corrispondente dal database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **EmailSubmissionPort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sull'invio di un'email per la consegna a uno o più destinatari all'interno del database;
  - *Operazioni*:
    `# insert(accountid: String, submission: EmailSubmission): String` - Prende in input due parametri, l'identificatore dell'account e un oggetto EmailSubmission da inserire, e restituisce una

stringa che rappresenta l'identificativo univoco della sottomissione di email inserita nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.

- **ThreadPort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sui thread all'interno del database;
  - *Operazioni*:
    `# get(id: String): Thread` - Prende in input un parametro che rappresenta l'identificativo univoco di un thread e restituisce l'oggetto Thread corrispondente. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# insert(account: String, id: Thread): String` - Prende in input due parametri, l'identificatore dell'account e un oggetto Thread da inserire, e restituisce una stringa che rappresenta l'identificativo univoco del thread inserito nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **ThreadUpdatePort**: porta in uscita che definisce una serie di metodi per l'accesso alle informazioni sugli aggiornamenti dei thread e per svolgere operazioni su di essi all'interno del database.
  - *Operazioni*:
    `# get(accountid: String, state: String): Update` - Prende in input due parametri, l'identificatore dell'account e lo stato corrente dei thread, e restituisce l'oggetto Update relativo alle modifiche apportate ai thread associati all'account specificato. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# getOf(accountid: String): Map<String, Update>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere gli aggiornamenti dei thread e restituisce una mappa che associa gli identificatori univoci dei thread agli oggetti Update corrispondenti. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# insert(accountid: String, oldstate: String, update: Update): String` - Prende in input tre parametri, l'identificatore dell'account, lo stato precedente dei thread e un oggetto Update rappresentante le modifiche apportate ai thread, e restituisce una stringa che rappresenta un identificativo univoco dell'operazione di aggiornamento dei thread inserita nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **UpdatePort**: porta in uscita che definisce una serie di metodi per gestire gli aggiornamenti nel sistema.
  - *Operazioni*:
    `# get(id: String): Update` - Prende in input un parametro che rappresenta l'identificativo univoco di un aggiornamento e restituisce l'oggetto Update corrispondente. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# getOf(accountid: String): Map<String, Update>` - Prende in input un parametro che rappresenta l'identificatore dell'account di cui si desiderano ottenere gli aggiornamenti e restituisce una mappa che associa gli identificatori univoci degli aggiornamenti agli oggetti Update corrispondenti. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# insert(id: String, update: Update): String` - Prende in input due parametri, l'identificativo univoco dell'aggiornamento e un oggetto Update da inserire, e restituisce una stringa che rappresenta l'identificativo univoco dell'aggiornamento inserito nel database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
    `# java delete(id: String): void` - Prende in input un parametro che rappresenta l'identificativo univoco di un aggiornamento e lo elimina dal database. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.
- **AttachmentPort**: porta in uscita che definisce una serie di metodi per svolgere operazioni sugli allegati all'interno del database;
  - *Operazioni*:
    `# get(id: String): byte[]` - Prende in input un parametro che rappresenta l'identificativo uni-

voco di un allegato e restituisce un array di byte che rappresenta i dati dell'allegato corrispondente. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
`insert(data: byte[]): String` - Prende in input un array di byte che rappresenta i dati di un allegato e inserisce l'allegato nel database. Restituisce una stringa che rappresenta l'identificativo univoco dell'allegato inserito. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia;
`delete(id: String): boolean` - Prende in input un parametro che rappresenta l'identificativo univoco di un allegato e elimina l'allegato corrispondente dal database. Restituisce true se l'operazione di eliminazione è avvenuta con successo, altrimenti restituisce false. L'implementazione concreta è lasciata alle classi che implementano questa interfaccia.

- **AccountRepository**: implementazione dell'interfaccia AccountPort che realizza i metodi necessari per operare con gli account degli utenti all'interno del database;
  - *Operazioni*:
    + `getId(username: String): String` Implementa il metodo getId dell'interfaccia AccountPort;
    + `getPassword(id: String): String` Implementa il metodo getPassword dell'interfaccia AccountPort.
- **StateRepository**: implementazione dell'interfaccia StatePort che realizza i metodi necessari per svolgere operazioni riguardanti lo stato dei client all'interno del database;
  - *Operazioni*:
    + `getId(username: String): String` Implementa il metodo getId dell'interfaccia AccountPort;
    + `getPassword(id: String): String` Implementa il metodo getPassword dell'interfaccia AccountPort.
- **IdentityRepository**: implementazione dell'interfaccia IdentityPort che realizza i metodi necessari per l'accesso alle informazioni sulle identità, nonché per svolgere operazioni su quest'ultime all'interno del database;
  - *Operazioni*:
    + `getOf(accountid: String): Identity[]` Implementa il metodo getOf dell'interfaccia IdentityPort;
- **EmailRepository**: implementazione dell'interfaccia EmailPort che realizza i metodi necessari per svolgere operazioni sulle email all'interno del database;
  - *Operazioni*:
    + `get(id: String): Email` Implementa il metodo get dell'interfaccia EmailPort;
    + `getOf(accountid: String): Map<String, Email>` Implementa il metodo getOf dell'interfaccia EmailPort;
    + `insert(accountid: String, email: Email): void` Implementa il metodo insert dell'interfaccia EmailPort;
    + `delete(id: String): void` Implementa il metodo delete dell'interfaccia EmailPort.
- **EmailChangesRepository**: implementazione dell'interfaccia EmailChangesPort che realizza i metodi necessari per svolgere operazioni sui cambiamenti avvenuti alle email all'interno del database;
- **MailboxChangesRepository**: implementazione dell'interfaccia MailboxChangesPort che realizza i metodi necessari per svolgere operazioni sui cambiamenti avvenuti alle cartelle all'interno del database;
- **MailboxRepository**: implementazione dell'interfaccia MailboxPort che realizza i metodi necessari per svolgere operazioni sulle caselle di posta all'interno del database;
- **EmailSubmissionRepository**: implementazione dell'interfaccia EmailSubmissionPort che realizza i metodi necessari per svolgere operazioni sull'invio di un'email per la consegna a uno o più destinatari all'interno del database;

- *Operazioni*:
  + `insert(accountid: String, email: EmailSubmission): String` Implementa il metodo insert dell'interfaccia EmailSubmissionPort. TO DO nel codice?
- **EmailUpdateRepository**: implementazione dell'interfaccia EmailUpdatePort che realizza i metodi necessari per svolgere operazioni sui cambiamenti avvenuti alle email all'interno del database;
  - *Operazioni*:
    + `get(accountid: String, state: String): Update` Implementa il metodo get dell'interfaccia EmailUpdatePort;
    + `getOf(accountid: String): Map<String, Update>` Implementa il metodo getOf dell'interfaccia EmailUpdatePort;
    + `delete(accountid: String): void` Implementa il metodo delete dell'interfaccia EmailUpdatePort. TO DO nel codice? è commentato
- **ThreadRepository**: implementazione dell'interfaccia ThreadPort che realizza i metodi necessari per svolgere operazioni sui thread all'interno del database;
- **AttachmentRepository**: implementazione dell'interfaccia AttachmentPort che realizza i metodi necessari per svolgere operazioni sugli allegati all'interno del database.
  - *Operazioni*:
    + `get(id: String): byte[]` Implementa il metodo get dell'interfaccia AttachmentPort;
    + `delete(id: String): boolean` Implementa il metodo delete dell'interfaccia AttachmentPort;
    + `insert(data: byte[]): String` Implementa il metodo delete dell'interfaccia AttachmentPort.

4.5) Database

Come già citato nella sezione **Tecnologie** del documento, il nostro prodotto utilizza RethinkDB come database NoSQL principale per la gestione dei dati. Il database viene inizializzato con la creazione delle collezioni richieste (account, email, mailbox...) e l'inserimento di dati di esempio. Successivamente, viene utilizzato per l'aggiunta di nuovi dati o la sostituzione di quelli esistenti.
Inoltre, per la gestione degli allegati abbiamo scelto di utilizzare una tecnologia diversa, affidandoci a MinIO. Questa scelta è stata fatta principalmente per dimostrare le potenzialità della nostra architettura che, operando tramite porte, ci permette di utilizzare sistemi di persistenza differenti con estrema facilità.

4.5.1) Scelta di RethinkDB

RethinkDB è stata la nostra scelta principale per molteplici motivi che rispecchiano le esigenze uniche del progetto: la necessità di un sistema altamente flessibile, scalabile e performante, in grado di adattarsi a diverse condizioni operative, che includono sia situazioni normali che di elevato carico e sovraccarico.

4.5.2) Funzionalità di RethinkDB

- **Modello di dati flessibile**: la natura NoSQL di questo database ci consente di modellare i dati in modo flessibile, senza vincoli rigidi di schema. Ciò significa che possiamo memorizzare email, cartelle, metadati e altri dati associati in un formato che si adatta alle esigenze specifiche del nostro sistema, consentendoci di gestire la complessità del nostro dominio in modo efficiente;
- **Scalabilità Orizzontale**: RethinkDB è progettato per scalare orizzontalmente, consentendo al nostro sistema di gestire volumi crescenti di dati e carichi di lavoro variabili. Questa funzionalità è fondamentale per assicurare che il sistema possa crescere in modo fluido con l'aumentare del carico, senza compromettere le prestazioni o la disponibilità del servizio;
- **Real time**: RethinkDB è stato progettato per supportare applicazioni che richiedono aggiornamenti in tempo reale dei dati grazie alla sua capacità di fornire un flusso continuo di cambiamenti ai dati attraverso i feed dei cambiamenti: questo lo rende particolarmente adatto per l'integrazione con un server di posta elettronica.

RethinkDB offre, inoltre, un potente sistema di query che semplifica l'accesso e la manipolazione dei dati, inclusi strumenti come subqueries e changefeed.

- **Subqueries**: sono query annidate all'interno di altre query e consentono agli sviluppatori di scrivere query più complesse e efficienti per soddisfare le esigenze specifiche delle loro applicazioni;
- **Changefeeds**: permettono agli sviluppatori di tracciare le modifiche nei dati e di ricevere notifiche istantanee quando avvengono cambiamenti nel database, facilitando lo sviluppo di applicazioni reattive.

4.5.3) Utilizzo di RethinkDb nel nostro progetto

In questa sezione andremo ad analizzare (come abbiamo fatto con tutte le tecnologie viste fino ad ora) un caso di utilizzo vero e proprio di RethinkDB nel nostro progetto. Come prima cosa, andremo a vedere la procedura necessaria per creare la connessione al database, quindi il collegamento tra il nostro server e il database. In secondo luogo andremo ad approfondire una classe Java all'interno del nostro progetto per mostrare le procedure di base per inserire o prelevare dati dal nostro db.

4.5.3.1) Creazione di una connessione, quindi collegamento al database

Per effettuare una connessione con il Database, vista la nostra architettura, abbiamo creato una classe apposita: `RethinkDBConnection.java`.

```
public class RethinkDBConnection {
  private Connection conn;

  public RethinkDBConnection(String host, Integer port, String db) {
    this.conn = RethinkDB.r.connection().hostname(host).port(port).connect().use(db);
  }

  @Provides
  public Connection provideConnection() {
    return conn;
  }
}
```

Questa classe viene istanziata nel metodo main, dove le vengono fornite tutte le informazioni per poter effettuare il collegamento al Database.

```
new RethinkDBConnection(
    System.getenv("RETHINKDB_HOST"),
    Integer.parseInt(System.getenv("RETHINKDB_PORT")),
    System.getenv("RETHINKDB_DB")
);
```

4.5.3.2) Operazioni sui dati del database

Eseguiti questi step preliminari, possiamo mostrare un utilizzo vero e proprio del database nella classe: `EmailRepository`. Qui troviamo il metodo `get` che data una stringa id, trova nel database l'oggetto Email serializzato sottoforma di json con quel determinato id e lo resistituisce al metodo che successivamente farà un cast per trasformarlo in un oggetto Email.

Di seguito il metodo in questione.

```
public class EmailRepository implements EmailPort {
  private final RethinkDB r = RethinkDB.r;
  // private final TypeReference<Map<String, Object>> stringObjectMap =
Types.mapOf(String.class, Object.class);
  private Connection conn;
  private Gson gson;

  @Inject
  EmailRepository(Connection conn, Gson gson) {
    this.conn = conn;
    this.gson = gson;
  }

  @Override
  public Email get(String id) {
    String res = r.table("email")
                  .get(id)
                  .toJson()
                  .run(conn)
                  .single()
                  .toString();
    return gson.fromJson(res, Email.class);
  }
```

4.5.4) Utilizzo di MinIO

MinIO è un sistema open-source di archiviazione di oggetti ideale per archiviare grandi quanittà di dati non strutturati (anche immagini, video e grandi backup sono inclusi). È compatibile con lo standard S3 (Simple Storage Service) di Amazon Web Services (AWS) ed è progettato per fornire una soluzione di storage di oggetti ad alte prestazioni. Gli oggetti in MinIO sono archiviati in modo distribuito su nodi multipli, consentendo una rapida accessibilità e un'alta disponibilità, questo fa si che le performance di MinIO consentano di supportare un carico di lavoro che i tradizionali sistemi di archiviazione di oggetti non possono supportare.

4.5.4.1) Implementazione di MinIO sul nostro progetto

In questa sezione andremo ad approfondire come abbiamo utilizzato MinIO all'interno del nostro progetto analizzando la classe `AttachmentRepository.java`.

In questa prima parte avviene la connessione a MinIO.

```
public class AttachmentRepository implements AttachmentPort {
  private MinioClient conn;
  private BucketName buck;

  @Inject
  AttachmentRepository(MinioClient conn, BucketName buck) {
    this.conn = conn;
    this.buck = buck;
  }
```

Si dichiara un campo `MinIO conn` per rappresentare la connessione al servizio MinIO. Questa connessione viene iniettata nel costruttore della classe attraverso l'annotazione @Inject di Guice. Si dichiara anche un campo che è un'istanza della classe `BucketName` per rappresentare il nome del bucket MinIO e l'hai iniettata anch'essa nel costruttore della classe.

In questa seconda parte avviene l'implementazione delle operazioni di base:

```
@Override
  public byte[] getAttachment(String id) {
    try {
      return conn.getObject(
        GetObjectArgs.builder()
          .bucket(buck.getName())
          .object(id)
          .build()).readAllBytes();
    } catch (Exception e) {
    }
    return null;
  }

  @Override
  public boolean deleteAttachment(String id) {
    try {
      conn.removeObject(
        RemoveObjectArgs.builder()
          .bucket(buck.getName())
          .object(id)
          .build());
      return true;
```

```
    } catch (Exception e) {
    }
    return false;
  }

  @Override
  public String insertAttachment(byte[] data) {
    try {
      var name = UUID.randomUUID().toString();
      conn.putObject(
        PutObjectArgs.builder()
          .bucket(buck.getName())
          .object(name)
          .stream(new ByteArrayInputStream(data), 0, -1)
          .build());
        return name;
    } catch (Exception e) {
    }
    return null;
  }
```

Il recupero di un allegato avviene tramite il metodo `getAttachment`, utilizzando conn.getObject con un oggetto GetObjectArgs.
Per inserire un nuovo allegato si utilizza il metodo `insertAttachment` che genera un nome univoco usando `UUID.randomUUID()` e successivamente usato `conn.putObject` con un oggetto `PutObjectArgs`.
Per rimuovere un allegato invece si utilizza il metodo `deleteAttachment`, che utilizza `conn.removeObject` con un oggetto `RemoveObjectArgs`.

4.5.5) Conclusioni

L'adozione di RethinkDB nel nostro server di posta elettronica rappresenta un elemento chiave nella nostra strategia di gestione dei dati. Grazie alla sua flessibilità, scalabilità e capacità di query avanzate, siamo in grado di offrire un servizio di posta elettronica affidabile, efficiente e altamente performante. Questo ci permette di ottenere i migliori risultati possibili durante i test di carico (stress test) richiesti dal proponente.
Inoltre l'uso di MinIO per la gestione degli allegati delle email non solo ci permette di dar prova dei vantaggi della nostra scelta architetturale, bensì offre anch'esso prestazioni elevate, scalabilità, affidabilità, flessibilità e sicurezza, contribuendo a ottimizzare il processo di gestione delle email e migliorando l'esperienza degli utenti finali.

5) Stato dei requisiti funzionali

Vengono di seguito riportati i requisiti funzionali individuati durante la fase di analisi. Per ognuno di essi vengono forniti:

- **Codice**: identificativo;
- **Tipo**: priorità;
- **Descrizione**;
- **Stato**: soddisfatto/non soddisfatto;
- **Riferimento**: classe del prodotto in cui il requisito è stato realizzato.

Per maggiori dettagli su Codice e Tipo si rimanda alla sezione `Requisiti` del documento `Analisi dei Requisiti v2.0.0`.

Codice	Tipo	Descrizione	Stato
R-001-F-2	Desiderabile	L'utente che utilizza un client di posta elettronica per interagire con il server deve autenticarsi all'interno del sistema.	Soddisfatto
R-002-F-2	Desiderabile	È necessario che il client fornisca all'interno della richiesta l'indirizzo email personale dell'utente per procedere con l'autenticazione.	Soddisfatto
R-003-F-2	Desiderabile	È necessario che il client fornisca all'interno della richiesta la password associata all'indirizzo email personale dell'utente per procedere con l'autenticazione.	Soddisfatto
R-004-F-2	Desiderabile	Se la fase di autenticazione è fallita allora è necessario che il client riceva dal server una risposta con eventuali dettagli che ne indicano il motivo.	Soddisfatto
R-005-F-1	Obbligatorio	Il client deve essere in grado di reperire la risorsa JMAP Session, contenente informazioni sulle capacità del server, dettagli sull'account dell'utente e le URL per le richieste API future, in modo da poter interagire con dati e servizi offerti dal server.	Soddisfatto
R-006-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "unkownCapability" in caso di esecuzione di una richiesta con proprietà "using" non supportata dal server.	Soddisfatto
R-007-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "notJSON" se il contenuto di una richiesta inviata al server non era application/json o se la richiesta non è stata interpretata dal server come I-JSON.	Soddisfatto
R-008-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "notRequest" se una richiesta JSON non ha corrisposto alla firma di tipo dell'oggetto di richiesta (Request).	Soddisfatto
R-009-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "limit" in caso di inserimento di una richiesta che supera uno dei limiti definiti sull'oggetto di capacità, come maxSizeRequest, maxCallsInRequest o maxCurrentRequests.	Soddisfatto
R-010-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "serverUnavailable" in caso di inserimento di una richiesta che necessita di alcune risorse interne del server momentaneamente non disponibili.	Soddisfatto
R-011-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "serverFail" in caso si verifichi un errore inaspettato o sconosciuto durante l'elaborazione di una sua richiesta dal server.	Soddisfatto
R-012-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "serverPartialFail" e proceda risincronizzando i dati in caso si verifichi un errore	Soddisfatto

		inaspettato o sconosciuto durante l'elaborazione di una sua richiesta dal server.	
R-013-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "unknownMethod" in caso di inserimento di una richiesta contenente un metodo non riconosciuto dal server.	Soddisfatto
R-014-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidArguments" se uno degli argomenti di un metodo fornito all'interno di una richiesta al server è di tipo errato, non valido o, nel caso in cui sia obbligatorio, è assente.	Soddisfatto
R-015-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidResultReference" se uno degli argomenti di un metodo fornito all'interno di una richiesta al server ha utilizzato un riferimento di risultato che non è stato possibile risolvere da parte del server.	Soddisfatto
R-016-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "forbidden" in caso utilizzi, all'interno di una richiesta al server, un metodo la cui esecuzione violerebbe una Access Control List (ACL) o un'altra policy di autorizzazione.	Soddisfatto
R-017-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "accountNotFound" se l'"accountID" fornito all'interno di una richiesta al server non corrisponde a un account valido.	Soddisfatto
R-018-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "accountNotSupportedByMethod" se all'interno di una richiesta al server è presente un metodo o tipo di dato non supportato dall'"accountID" fornito.	Soddisfatto
R-019-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "accountReadOnly" se all'interno di una richiesta al server è presente un metodo che tenta di modificare lo stato nonostante l'account sia in sola lettura.	Soddisfatto
R-020-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "cannotCalculateChanges" se, in seguito all'inserimento di una richiesta, il server non possa calcolare le modifiche dello stato dalla stringa di stato fornita dal client.	Soddisfatto
R-021-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "overQuota" se una richiesta inserita nel server richiede la creazione di oggetti che per dimensione o quantità superano il limite imposto dal server.	Soddisfatto
R-022-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "notFound" se una richiesta inserita nel server fornisce degli ID che non possono essere trovati.	Soddisfatto
R-023-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "willDestroy" se ha richiesto che un oggetto fosse sia aggiornato che distrutto all'interno della stessa richiesta al server.	Soddisfatto
R-024-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "tooLarge" se una richiesta inserita nel server richiede la creazione di un oggetto che supera il limite definito dal server per la dimensione massima per un oggetto di quel tipo.	Soddisfatto
R-025-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "rateLimit" se una richiesta inserita nel server comporta la creazione di un oggetto per cui sono stati creati troppi oggetti quel tipo di recente, raggiungendo un limite di frequenza definito dal server.	Soddisfatto

R-026-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidPatch" se una richiesta inserita nel server fornisce un PatchObject non valido per modificare il record.	Soddisfatto
R-027-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidProperties" se una richiesta inserita nel server fornisce un record non valido.	Soddisfatto
R-028-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "singleton" se una richiesta inserita nel server tentasse di agire erroneamente su un tipo singleton.	Soddisfatto
R-029-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "requestTooLarge" se una richiesta inserita nel server contiene un numero di azioni che supera il massimo che il server è disposto a elaborare in una singola chiamata di metodo interna alla richiesta.	Soddisfatto
R-030-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "stateMismatch" se una richiesta inserita nel server contiene un argomento ifInState e questo non corrisponde allo stato attuale.	Soddisfatto
R-031-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "blobNotFound" se una richiesta inserita nel server contiene almeno un ID blob fornito per una parte del corpo dell'email che non esiste.	Soddisfatto
R-032-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "tooManyKeywords" se una richiesta inserita nel server modifica un numero di parole chiave dell'email superiore al limite massimo definito dal server.	Soddisfatto
R-033-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "tooManyMailboxes" se una richiesta inserita nel server modifica un numero di cartelle a cui appartiene l'email superiore al limite massimo definito dal server.	Soddisfatto
R-034-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "alreadyExists" se una richiesta inserita in un server che vieta i duplicati contiene un record già esistente nell'account di destinazione.	Soddisfatto
R-035-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "fromAccountNotFound" se una richiesta inserita nel server contiene un fromAccountId che non corrisponde a nessun account valido.	Soddisfatto
R-036-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "fromAccountNotSupportedByMethod" se una richiesta inserita nel server contiene un fromAccountId che non supporta un tipo di dato utilizzato.	Soddisfatto
R-037-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "anchorNotFound" se una richiesta inserita nel server contiene un argomento di ancoraggio che non è stato trovato nei risultati della query.	Soddisfatto
R-038-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "unsupportedSort" se una richiesta inserita nel server presenta una clausola di ordinamento non supportata o un metodo di collezione non riconosciuto dal server.	Soddisfatto
R-039-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "unsupportedFilter" se una richiesta inserita nel server contiene un filtro che il server non è grado di elaborare.	Soddisfatto
R-040-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "tooManyChanges" se una richiesta inserita nel server contiene un	Soddisfatto

		numero di modifiche superiore all'argomento maxChanges inserito del client.	
R-041-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "mailboxHasChild" se una richiesta inserita nel server desidera rimuovere una cartella(Mailbox) che ha ancora almeno una cartella figlia.	Soddisfatto
R-042-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "mailboxHasEmail" se una richiesta inserita nel server, con l'argomento onDestroyRemoveEmails impostato su false, desidera rimuovere una cartella(Mailbox) che ha al suo interno almeno una email.	Soddisfatto
R-043-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidEmail" se una richiesta inserita nel server contiene un'email da inviare non valida.	Soddisfatto
R-044-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "tooManyRecipients" se una richiesta inserita nel server contiene un envelope (insieme di destinatari) che ha più destinatari di quanti il server consenta.	Soddisfatto
R-045-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "noRecipients" se una richiesta inserita nel server contiene un envelope (insieme di destinatari) che non presenta alcun destinatario.	Soddisfatto
R-046-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "invalidRecipients" se una richiesta inserita nel server contiene un envelope (insieme di destinatari) con almeno un indirizzo email destinatario non valido.	Soddisfatto
R-047-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "forbiddenMailFrom" se una richiesta è inserita in un server che non consente all'utente di inviare un messaggio con quel indirizzo mittente nell'envelope (From address).	Soddisfatto
R-048-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "forbiddenFrom" se una richiesta è inserita in un server che non consente all'utente di inviare un messaggio con il campo di intestazione From del messaggio da inviare.	Soddisfatto
R-049-F-1	Obbligatorio	È necessario che il client riceva in risposta l'errore "forbiddenToSend" se una richiesta è inserita in un server che non consente all'utente di inviare un messaggio in quel momento.	Soddisfatto
R-050-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di inviare email.	Soddisfatto
R-051-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie all'invio di una email.	Soddisfatto
R-052-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie all'invio di una email, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-053-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le proprietà dell'oggetto Email da creare (cartelle in cui è contenuta, mittente, destinatari, oggetto, corpo del messaggio ed altri dettagli definiti dall'RFC5322).	Soddisfatto
R-054-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le proprietà dell'oggetto EmailSubmission da creare (l'identificativo	Soddisfatto

		dell'email creata in precedenza ed ora da inviare, le informazioni necessarie per l'invio ed altri dettagli).	
R-055-F-1	Obbligatorio	È possibile che il client inserisca all'interno della richiesta eventuali azioni da compiere in seguito al corretto invio dell'email.	Soddisfatto
R-056-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di invio dell'email con relativi parametri (come il nuovo stato, gli oggetti creati ed eventuali errori).	Soddisfatto
R-057-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di ricevere email e visualizzarne il dettaglio.	Soddisfatto
R-058-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla ricezione di una email.	Soddisfatto
R-059-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla ricezione di una email, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-060-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificativo delle email da ricevere	Soddisfatto
R-061-F-1	Obbligatorio	È possibile che il client inserisca all'interno della richiesta le proprietà specifiche delle email che è interessato a ricevere	Soddisfatto
R-062-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di ricezione dell'email con relativi parametri (come lo stato corrente dei dati di tipo Email sul server, la lista delle email richieste ed eventuali errori)	Soddisfatto
R-063-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di eliminare email.	Soddisfatto
R-064-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie all'eliminazione di una email.	Soddisfatto
R-065-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie all'eliminazione di una email, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-066-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta la lista degli identificativi delle email da eliminare.	Soddisfatto
R-067-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di eliminazione delle email con relativi parametri (come il nuovo stato, gli identificativi degli oggetti eliminati ed eventuali errori)	Soddisfatto
R-068-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di ricevere cartelle e visualizzarne il dettaglio.	Soddisfatto
R-069-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla ricezione di una cartella.	Soddisfatto
R-070-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla ricezione di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo	Soddisfatto

		dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	
R-071-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificativo delle cartelle da ricevere	Soddisfatto
R-072-F-1	Obbligatorio	È possibile che il client inserisca all'interno della richiesta le proprietà specifiche delle cartelle che è interessato a ricevere	Soddisfatto
R-073-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di ricezione della cartella con relativi parametri (come lo stato corrente dei dati di tipo Mailbox sul server, la lista delle cartelle richieste ed eventuali errori)	Soddisfatto
R-074-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di creare cartelle.	Soddisfatto
R-075-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla creazione di una cartella.	Soddisfatto
R-076-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla creazione di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-077-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le proprietà dell'oggetto Mailbox da creare (nome, eventuale genitore, ruolo ed altri dettagli).	Soddisfatto
R-078-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di creazione della cartella con relativi parametri (come lo stato corrente del server, la lista delle cartelle create ed eventuali errori)	Soddisfatto
R-079-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di modificare cartelle esistenti.	Soddisfatto
R-080-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla modifica di una cartella.	Soddisfatto
R-081-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla modifica di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-082-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare all'oggetto Mailbox che l'utente desidera modificare.	Soddisfatto
R-083-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di modifica della cartella con relativi parametri (come lo stato corrente del server, la lista delle cartelle modificate ed eventuali errori)	Soddisfatto
R-084-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di eliminare cartelle esistenti.	Soddisfatto
R-085-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla eliminazione di una cartella.	Soddisfatto

R-086-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla eliminazione di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-087-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta la lista degli identificativi delle cartelle da eliminare.	Soddisfatto
R-088-F-1	Obbligatorio	È necessario che il client specifichi all'interno della richiesta il comportamento desiderato da parte del server quando si cerca di eliminare una cartella che contiene ancora delle email.	Soddisfatto
R-089-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di eliminazione della cartella con relativi parametri (come lo stato corrente del server, la lista degli identificativi delle cartelle eliminate ed eventuali errori)	Soddisfatto
R-090-F-1	Obbligatorio	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di gestire i contenuti di una cartella.	Soddisfatto
R-091-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla gestione dei contenuti di una cartella.	Soddisfatto
R-092-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla gestione dei contenuti di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-093-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare agli oggetti Email che l'utente desidera aggiungere ad una o più cartelle.	Soddisfatto
R-094-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare agli oggetti Email che l'utente desidera rimuovere da una o più cartelle.	Soddisfatto
R-095-F-1	Obbligatorio	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare agli oggetti Email che l'utente desidera spostare da una o più cartelle ad una o più cartelle.	Soddisfatto
R-096-F-1	Obbligatorio	È necessario che il client riceva una risposta che contiene l'esito delle operazioni di gestione dei contenuti di una cartella con relativi parametri (come lo stato corrente del server, la lista delle email modificate ed eventuali errori)	Soddisfatto
R-097-F-2	Desiderabile	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di condividere le sue cartelle con altri utenti.	Non soddisfatto
R-098-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla creazione della condivisione di una cartella.	Non soddisfatto
R-099-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla creazione della condivisione di una cartella, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Non soddisfatto
R-100-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le proprietà dell'oggetto Principal da creare (nome, tipo, descrizione, gli	Non soddisfatto

		identificativi degli account a cui vogliamo condividere le cartelle e altri dettagli).	
R-101-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare all'oggetto Mailbox che l'utente desidera condividere, specificando quali diritti hanno su quest'ultimo i membri del principale a cui si sta condividendo.	Non soddisfatto
R-102-F-2	Desiderabile	È necessario che il client riceva una risposta che contiene l'esito delle operazioni di creazione della condivisione di una cartella con relativi parametri (come lo stato corrente dei dati sul server, la lista dei principali creati, la lista delle cartella modificate ed eventuali errori)	Non soddisfatto
R-103-F-2	Desiderabile	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di modificare principali esistenti.	Non soddisfatto
R-104-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla modifica di un principale.	Non soddisfatto
R-105-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla modifica di un principale, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Non soddisfatto
R-106-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare all'oggetto Principal che l'utente desidera modificare.	Non soddisfatto
R-107-F-2	Desiderabile	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di modifica del principale con relativi parametri (come lo stato corrente del server, la lista dei principali modificati ed eventuali errori)	Non soddisfatto
R-108-F-2	Desiderabile	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di eliminare principali esistenti.	Non soddisfatto
R-109-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla eliminazione di un principale.	Non soddisfatto
R-110-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla eliminazione di un principale, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Non soddisfatto
R-111-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta la lista degli identificativi dei principali da eliminare.	Non soddisfatto
R-112-F-2	Desiderabile	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di eliminazione del principale con relativi parametri (come lo stato corrente del server, la lista degli identificativi dei principali eliminati ed eventuali errori)	Non soddisfatto
R-113-F-2	Desiderabile	L'utente che utilizza un client di posta elettronica per interagire con il server deve avere la possibilità di modificare la condivisione di una cartella (compresa l'eliminazione di quest'ultima).	Non soddisfatto
R-114-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla modifica della condivisione di una cartella (compresa l'eliminazione di quest'ultima).	Non soddisfatto

R-115-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla modifica della condivisione di una cartella (compresa l'eliminazione di quest'ultima), con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Non soddisfatto
R-116-F-2	Desiderabile	È necessario che il client inserisca all'interno della richiesta le modifiche da apportare all'oggetto Mailbox di cui l'utente desidera modificare/eliminare la condivisione, specificando i nuovi diritti che hanno su quest'ultimo i membri del principale a cui si sta condividendo.	Non soddisfatto
R-117-F-2	Desiderabile	È necessario che il client riceva una risposta che contiene l'esito dell'operazione di modifica della condivisione di una cartella (compresa l'eliminazione di quest'ultima) con relativi parametri (come lo stato corrente del server, la lista degli identificativi delle cartelle modificate ed eventuali errori)	Non soddisfatto
R-118-F-3	Opzionale	Un client di posta elettronica utilizzato da un utente per interagire con il server deve avere la possibilità di mantenersi sincronizzato con gli ultimi aggiornamenti per quanto riguarda le email.	Soddisfatto
R-119-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla sincronizzazione delle email.	Soddisfatto
R-120-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla sincronizzazione delle email, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto
R-121-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta il suo stato corrente per quanto riguarda le email, con lo scopo di sincronizzarsi.	Soddisfatto
R-122-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta il numero massimo di identificatori di email che desidera ricevere come risposta, con lo scopo di sincronizzarsi.	Soddisfatto
R-123-F-3	Opzionale	È necessario che il client riceva una risposta che contiene le informazioni di cui ha bisogno per sincronizzarsi (come lo stato corrente del server, un flag booleano che indica se ci sono ulteriori cambiamenti nel server relativi ai dati di tipo Email, oltre a quelli già restituiti nella risposta corrente, e la lista delle email da creare/modificare/eliminare per sincronizzarsi)	Soddisfatto
R-124-F-3	Opzionale	Un client di posta elettronica utilizzato da un utente per interagire con il server deve avere la possibilità di mantenersi sincronizzato con gli ultimi aggiornamenti per quanto riguarda le cartelle.	Soddisfatto
R-125-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta l'identificazione delle capacità necessarie alla sincronizzazione delle cartelle.	Soddisfatto
R-126-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta le chiamate di metodo necessarie alla sincronizzazione delle cartelle, con relativi parametri (sia quelli comuni, come l'identificativo dell'account da utilizzare, sia quelli specifici del caso) e un identificatore univoco associato.	Soddisfatto

R-127-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta il suo stato corrente per quanto riguarda le cartelle, con lo scopo di sincronizzarsi.	Soddisfatto
R-128-F-3	Opzionale	È necessario che il client inserisca all'interno della richiesta il numero massimo di identificatori di cartelle che desidera ricevere come risposta, con lo scopo di sincronizzarsi.	Soddisfatto
R-129-F-3	Opzionale	È necessario che il client riceva una risposta che contiene le informazioni di cui ha bisogno per sincronizzarsi (come lo stato corrente del server, un flag booleano che indica se ci sono ulteriori cambiamenti nel server relativi ai dati di tipo Mailbox, oltre a quelli già restituiti nella risposta corrente, e la lista delle cartelle da creare/modificare/eliminare per sincronizzarsi)	Soddisfatto

Tabella 9: Stato dei requisiti funzionali

5.1) Grafici riassuntivi

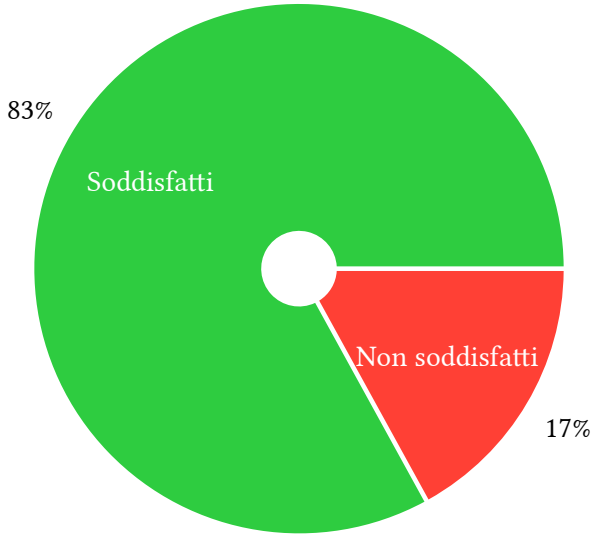

Figura 7: Stato dei requisiti funzionali totali

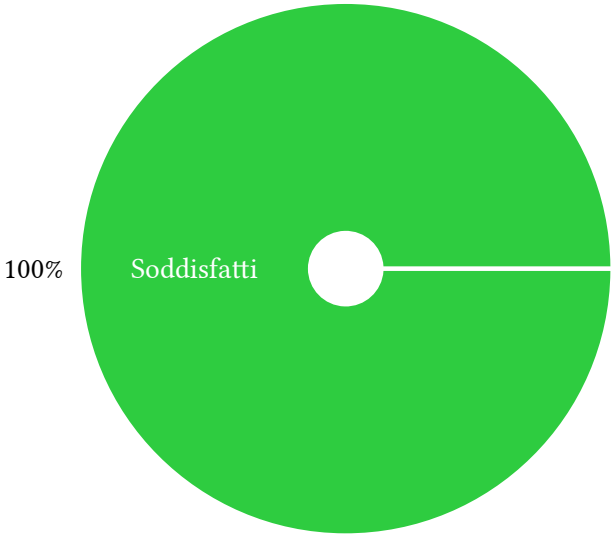

Figura 8: Stato dei requisiti funzionali obbligatori

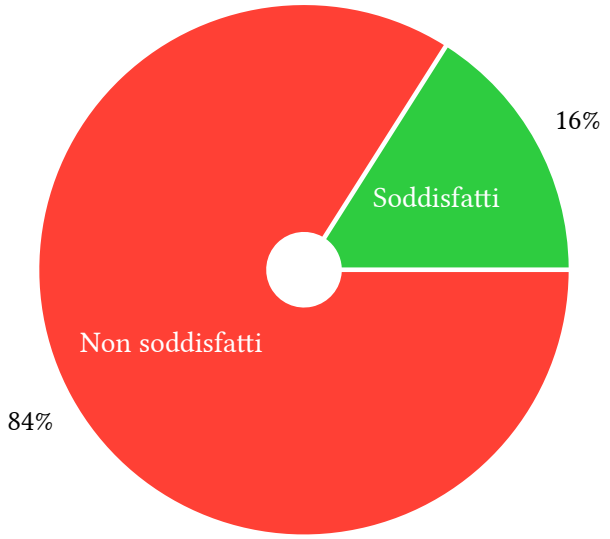

Figura 9: Stato dei requisiti funzionali desiderabili

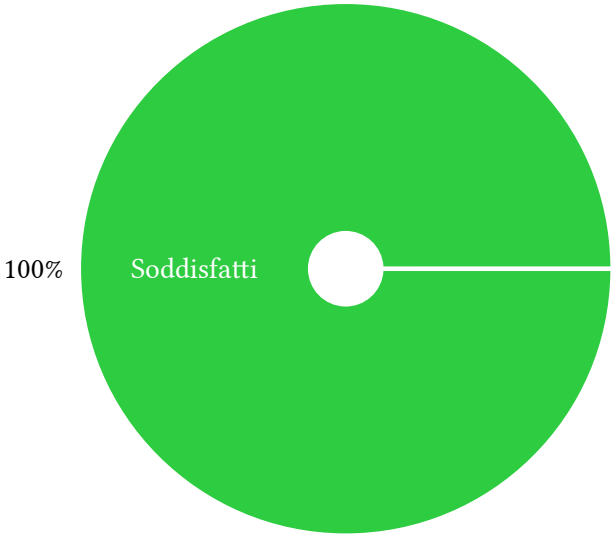

Figura 10: Stato dei requisiti funzionali opzionali