



Spendly: Documentazione Tecnica

Amin Borqal, Loris Iacoban, Diego Rossi

26 febbraio 2025

Indice

1	Iterazione 0	3
1.1	Introduzione	3
1.2	Requisiti	5
1.3	Casi d'Uso	6
1.4	Architettura del Sistema	8
1.4.1	Model	9
1.4.2	View	9
1.4.3	Controller	10
1.4.4	Flusso delle Richieste	10
1.5	Priorità casi d'uso	10
1.6	Topologia del Sistema	13
1.6.1	Gestione delle comunicazioni	13
1.6.2	Caratteristiche dell'architettura	14
1.7	Strumenti Utilizzati	15
2	Iterazione 1	18
2.1	Casi d'Uso Implementati	18
2.1.1	UC1: Login	18
2.1.2	UC2: Registrazione	18
2.1.3	UC3: Logout	19
2.1.4	UC7: Creazione Gruppo	19
2.1.5	UC8: Invita Membri	20
2.1.6	UC9: Elimina Membri	20
2.1.7	UC10: Modifica Membri	20
2.1.8	UC11: Eliminazione Gruppo	21
2.1.9	UC12: Accesso a un Gruppo	21
2.2	Diagramma delle Interfacce per Budget e Risparmio	22
2.3	UML Class Diagram per tipi di dato	23
2.4	UML Component Diagram	25
2.5	UML Deployment Diagram	26
2.6	Testing	28
2.6.1	Analisi Statica - CodMR	28
2.6.2	Analisi Dinamica - JUnit	29
2.6.3	API Esposte	31

3	Iterazione 2	37
3.1	Casi d'Uso Implementati	37
3.1.1	UC4: Creazione Alert di Gruppo	37
3.1.2	UC6: Eliminazione Alert	37
3.1.3	UC13: Inserimento Spesa	38
3.1.4	UC14: Eliminazione Spesa	38
3.1.5	UC15: Modifica Spesa	39
3.1.6	UC16: Visualizzazione Spese	39
3.1.7	UC17: Ricalcolo Debiti	39
3.2	UML Class Diagram per tipi di dato	41
3.3	UML Component Diagram	44
3.4	UML Deployment Diagram	46
3.5	Testing	47
3.5.1	Analisi Statica - CodeMR	47
3.5.2	API Costi	48
3.6	Algoritmo ottimizza debiti	52
3.6.1	PseudoCodice	52
3.6.2	Descrizione	54
3.6.3	Complessita	55
4	Iterazione 3	56
4.1	Introduzione	56
4.2	Casi d'Uso	56
4.2.1	Iterazione 3	57
4.2.2	UC24: Inserimento Spesa Personale	57
4.2.3	UC25: Eliminazione Spesa Personale	57
4.2.4	UC26: Modifica Spesa Personale	58
4.2.5	UC27: Visualizzazione Spese Personali	58
4.2.6	UC34: Pagamento Spesa Personale	58
4.2.7	UC28: Aggiunta Denaro al Budget	59
4.2.8	UC29: Visualizzazione Budget	59
4.2.9	UC30: Creazione Risparmio	59
4.2.10	UC31: Visualizzazione Risparmio	60
4.2.11	UC32: Modifica Risparmio	60
4.2.12	UC33: Eliminazione Risparmio	60
4.3	Diagramma delle Interfacce	61
4.4	UML Class Diagram per tipi di dato	62
4.5	UML Component Diagram	63

4.6	UML Deployment Diagram	64
-----	----------------------------------	----

Elenco delle figure

1	Diagramma casi d'uso	6
2	M-V-C Pattern	8
3	Diagramma dell'architettura della webapp Spendly	13
4	UML Deployment Diagram	27
5	Complexity	28
6	Coupling	28
7	Problemi classi	29
8	Struttura dei package	29
9	Test per la classe Group	30
10	Test per la classe GroupService	30
11	Test per la classe GroupController	31
12	Risultato API Registrazione	32
13	Risultato API Login	33
14	Risultato API Creazione Gruppo	34
15	Risultato API Inserimento utente	35
16	Risultato API Visualizzazione Gruppi	36
17	UML Deployment Diagram	46
18	Complexity	47
19	Coupling	47
20	Problemi classi	48
21	Struttura dei package	48
22	Risultato API Aggiunta Costo	50
23	Risultato API Eliminazione Costo	50
24	Risultato API Visualizzazione Costi Utente	51
25	Risultato API Visualizzazione Costi Gruppo	52
26	Funzione CreaDizionario	52
27	Funzione CalcolaDebiti	53
28	Diagramma dei casi d'uso aggiornato per riflettere l'integrazione relativa alla gestione del risparmio.	56
29	UML Deployment Diagram	64

Elenco delle tabelle

1	Coda alta priorità	11
2	Coda media priorità	12

3	Coda bassa priorità	12
4	Iterazione1	18
5	Iterazione2	37
6	Iterazione 3 - Casi d'Uso	57

1 Iterazione 0

1.1 Introduzione

Spendly è una web app progettata per semplificare la gestione delle spese personali e di gruppo, offrendo funzionalità che permettono a singoli utenti, famiglie o gruppi di organizzarsi in modo efficiente.

Con Spendly, è possibile:

- Creare gruppi di utenti, dove ciascun membro può registrare le spese effettuate per conto degli altri. L'applicazione calcolerà automaticamente il numero minimo di transazioni necessarie per saldare i debiti all'interno del gruppo.
- Impostare avvisi personalizzati per monitorare il livello di spesa del gruppo, fornendo notifiche utili per evitare superamenti del budget condiviso.

A livello individuale, ogni utente può:

- Visualizzare, registrare e gestire le proprie spese in modo autonomo.
- Creare e monitorare un budget di risparmio per pianificare meglio le proprie finanze personali.

Grazie a queste funzionalità, Spendly rende la gestione economica più semplice, trasparente ed efficace, sia per singoli utenti che per gruppi.

Perché scegliere Spendly?

Spendly è strutturata attorno a una serie di funzionalità che coprono ogni aspetto della gestione delle spese. Queste funzionalità sono state sviluppate con l'obiettivo di offrire la massima flessibilità, sia per gli utenti individuali che per i gruppi di condivisione spese. Spendly è più di una semplice applicazione di gestione delle spese. È una soluzione completa che integra funzionalità avanzate, come il calcolo automatico dei debiti e la gestione centralizzata delle spese. Questo la rende ideale per ogni tipo di utente, dai gruppi di amici che vogliono semplificare la divisione delle spese, alle famiglie che desiderano tenere sotto controllo i propri budget. Con una piattaforma sicura, flessibile e facile da usare, Spendly trasforma il modo in cui le persone gestiscono i

loro soldi, riducendo lo stress finanziario e migliorando la trasparenza nelle relazioni economiche. Spendly è lo strumento perfetto per chi desidera una gestione semplice, efficace e organizzata delle spese. Con funzionalità pensate per rispondere alle esigenze di utenti individuali e gruppi, questa web-app rappresenta una soluzione innovativa e accessibile per migliorare la vita quotidiana di chiunque voglia un controllo totale sulle proprie finanze.

1.2 Requisiti

Il sistema deve soddisfare i seguenti requisiti funzionali e non funzionali:

- **Requisiti Funzionali:**

- Gli utenti devono poter registrarsi e autenticarsi al sistema.
- Gli utenti devono poter aggiungere, modificare e cancellare le proprie spese.
- Gli utenti devono poter creare gruppi e gestire le spese condivise.
- Il sistema deve calcolare automaticamente il bilancio dei debiti tra i membri del gruppo.
- Il sistema deve inviare notifiche o alert agli utenti quando il loro budget per una determinata categoria di spesa viene superato.
- Gli utenti devono poter impostare budget personalizzati per categorie specifiche (es. alimentari, trasporti, svago, ecc.).

- **Requisiti Non Funzionali:**

- L'applicazione deve essere intuitiva e facile da usare.
- Il sistema deve garantire la sicurezza dei dati personali degli utenti mediante crittografia e autenticazione sicura.
- Le transazioni e le modifiche devono essere sincronizzate in tempo reale tra tutti i dispositivi degli utenti.
- L'applicazione deve essere accessibile da diversi dispositivi (smartphone, tablet, desktop).
- Il sistema deve garantire elevate prestazioni, assicurando una risposta rapida alle richieste degli utenti.

1.3 Casi d'Uso



Figura 1: Diagramma casi d'uso

Di seguito sono riportati i casi d'uso in figura:

- **UC1 - Login:** Come utente, voglio poter accedere al mio account per gestire le mie spese.
- **UC2 - Registrazione:** Come nuovo utente, voglio potermi registrare al sistema per iniziare a usare la web-app.
- **UC3 - Logout:** Voglio poter terminare la mia sessione.

- **UC4 - Crea alert di gruppo:** Come utente amministratore di un gruppo spesa voglio poter inserire un alert, dove un alert è un avviso che ci permette di avvisare se si sta raggiungendo una soglia limite di spesa.
- **UC5 - Modifica alert:** Voglio poter modificare i valori dell'alert.
- **UC6 - Elimina alert:** Voglio poter eliminare l'alert.
- **UC7 - Crea gruppo:** Voglio poter creare un gruppo di condivisione spese.
- **UC8 - Invita membri:** Voglio come amministratore invitare membri nel gruppo di spesa.
- **UC9 - Elimina membri:** Voglio come amministratore poter eliminare membri del gruppo di spesa.
- **UC10 - Modifica membri:** Voglio come amministratore modificare i membri nel gruppo di spesa.
- **UC11 - Elimina gruppo:** Voglio poter eliminare un gruppo di condivisione spese.
- **UC12 - Accedi gruppo:** Voglio poter accedere ad un gruppo di condivisione spese.
- **UC13 - Inserisci spesa:** Voglio poter inserire una spesa di gruppo.
- **UC14 - Elimina spesa:** Voglio poter eliminare una spesa di gruppo.
- **UC15 - Modifica spesa:** Voglio poter modificare una spesa di gruppo.
- **UC16 - Visualizza spese:** Voglio poter visualizzare le spesa di gruppo.
- **UC17 - Ricalcolo debiti:** Voglio poter calcolare i debiti.
- **UC18 - Inserisci spesa personale:** Voglio poter inserire una spesa personali.

- **UC19 - Elimina spesa personale:** Voglio poter eliminare una spesa personali.
- **UC20 - Modifica spesa personale:** Voglio poter modificare una spesa personali.
- **UC21 - Visualizza spesa personale:** Voglio poter visualizzare le spesa personali.
- **UC22 - Aggiungi denaro a budget:** Voglio poter inserire un budget di risparmio.
- **UC23 - Togli denaro da budget:** Voglio poter eliminare un budget di risparmio.
- **UC25 - Visualizza budget:** Voglio poter visualizzare i budget di risparmio.

1.4 Architettura del Sistema

Il sistema segue il pattern architetturale **Model-View-Controller (MVC)**, un'architettura software che separa la logica di presentazione, la logica di business e la gestione dei dati. Questa separazione consente una maggiore modularità e facilita la manutenzione e l'estendibilità del sistema. Le tre componenti principali sono:

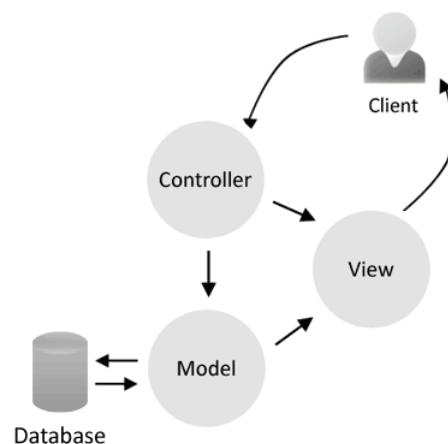


Figura 2: M-V-C Pattern

1.4.1 Model

Il **Model** rappresenta la logica di business e la gestione dei dati. In questo sistema:

- I dati sono memorizzati in un database relazionale **PostgreSQL**, scelto per la sua affidabilità e scalabilità.
- Il backend, implementato in **Spring Boot**, gestisce le operazioni sui dati attraverso un'architettura basata su servizi.
- I principali servizi includono:
 - **Servizio Utente**: gestione della registrazione, autenticazione (tramite **JWT**) e gestione del profilo utente.
 - **Servizio Gestione Spese**: esecuzione delle operazioni CRUD sulle spese degli utenti.
 - **Servizio Gruppi**: creazione e gestione dei gruppi di utenti.
- L'interazione con il database avviene tramite il **Data Access Layer**, basato su JPA e Hibernate.

1.4.2 View

La **View** è responsabile della presentazione dei dati all'utente e della gestione dell'interazione con il sistema. Nel nostro caso:

- Il frontend è sviluppato utilizzando **Vue.js**, un framework JavaScript progressivo che permette lo sviluppo di interfacce utente reattive e modulari.
- Le chiamate API al backend vengono gestite tramite **Axios**, consentendo un flusso di dati asincrono tra client e server.
- L'autenticazione degli utenti è gestita tramite token **JWT**, che vengono memorizzati e inviati nelle richieste HTTP per garantire la sicurezza.

1.4.3 Controller

Il **Controller** funge da intermediario tra il Model e la View, gestendo le richieste utente e applicando la logica di business. In questo sistema:

- Il backend espone un'API **RESTful** tramite Spring Boot, con endpoint per la gestione di utenti, spese e gruppi.
- Ogni richiesta HTTP viene elaborata da un controller dedicato, che interagisce con i servizi del Model per recuperare o aggiornare i dati.
- Le risposte sono restituite in formato JSON, consentendo una comunicazione fluida con il frontend.

1.4.4 Flusso delle Richieste

Il flusso di una tipica richiesta utente segue questi passi:

1. L'utente interagisce con il frontend Vue.js tramite un'interfaccia web.
2. Un'azione (es. login, aggiunta di una spesa) genera una richiesta HTTP verso il backend.
3. Il **Controller** riceve la richiesta e invoca il servizio appropriato.
4. Il **Model** esegue la logica di business e interagisce con il database per recuperare o aggiornare i dati.
5. La risposta viene inviata al frontend in formato JSON.
6. Il frontend aggiorna dinamicamente l'interfaccia utente con i nuovi dati.

Questa architettura garantisce una chiara separazione delle responsabilità, facilitando lo sviluppo, la scalabilità e la manutenzione del sistema.

1.5 Priorità casi d'uso

I casi d'uso possono essere suddivisi in tre categorie, a seconda della loro priorità nel processo di sviluppo:

- **Coda ad alta priorità:** Contiene i requisiti essenziali per il corretto funzionamento dell'applicativo. Questi includono la creazione dei profili utente, la creazione di nuovi gruppi spese, con le funzionalità annesse (invita, elimina membri ecc ecc) ed infine la gestione spesa con l'algoritmo di calcolo debiti.
- **Coda a media priorità:** Questa coda include funzionalità di supporto, principalmente orientate alla gestione spese personali.
- **Coda a bassa priorità:** Contiene le funzionalità meno rilevanti ossia la tematica del budget, per le quali non è prevista una loro implementazione immediata. Tuttavia, potrebbero essere implementate in futuro, a seconda dell'andamento del progetto.

ID	Titolo
UC1	Login
UC2	Registrazione
UC3	Logout
UC8	Invita membri
UC9	Elimina membri
UC10	Modifica membri
UC11	Elimina gruppo
UC12	Accedi gruppo
UC13	Inserisci spesa
UC14	Elimina spesa
UC15	Modifica spese
UC16	Visualizza spese
UC17	Ricalcolo debiti

Tabella 1: Coda alta priorità

ID	Titolo
UC4	Crea alert di gruppo
UC5	Modica alert
UC6	Elimina alert
UC18	Inserisci spesa personale
UC19	Elimina spesa personale
UC20	Modica spesa personale
UC21	Visualizza spesa personale

Tabella 2: Coda media priorità

ID	Titolo
UC22	Inserisci budget
UC23	Elimina budget
UC24	Modica budget
UC25	Visualizza budget

Tabella 3: Coda bassa priorità

Perché si dividono in code di priorità i casi d'uso?

In quanto lo sviluppo del progetto viene sviluppato tramite il processo Agile Model-Driven Development.

AMDD combina i principi della modellazione guidata dallo sviluppo con la flessibilità dei metodi agili, che a differenza dello sviluppo tradizionale, in cui tutta l'analisi viene completata prima di iniziare la programmazione, prevede una modellazione iniziale leggera e iterativa, concentrata solo sugli aspetti essenziali per l'iterazione corrente.

Ad ogni ciclo di sviluppo, il team seleziona un numero limitato di casi d'uso dalla coda di priorità più alta, lavorando su di essi fino a ottenere una funzionalità completa e testata. Questo approccio:

- Riduce il rischio affrontando le funzionalità critiche prima possibile.
- Favorisce il feedback continuo, permettendo agli stakeholder di validare il progresso e suggerire miglioramenti.
- Adatta lo sviluppo alle necessità reali, permettendo di rivalutare le priorità in base all'andamento del progetto.

Grazie a questo metodo, il sistema viene costruito in modo solido e incrementale, con la possibilità di aggiungere nuove funzionalità man mano che il progetto evolve.

1.6 Topologia del Sistema

La topologia del sistema è rappresentata nel seguente schema:

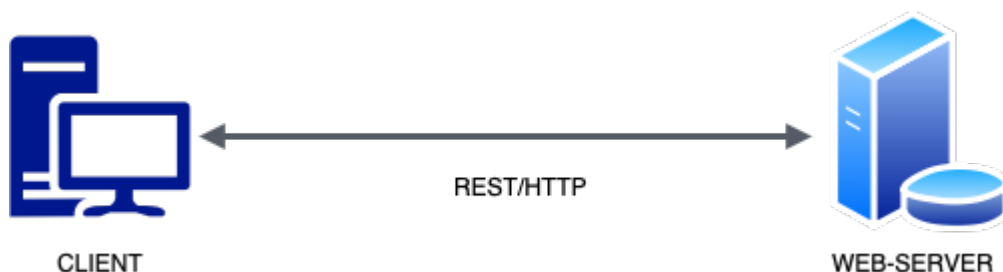


Figura 3: Diagramma dell'architettura della webapp Spendly

Il sistema è basato su un'architettura **Client-Server**, in cui i client interagiscono con un **web server** sviluppato utilizzando il framework **Spring Boot**. I client possono essere:

- **Applicazione Web:** gli utenti accedono a Spendly tramite un browser, comunicando con il server attraverso **API REST/HTTP**.
- **Applicazione Mobile:** gli utenti interagiscono con il sistema tramite un'app mobile che utilizza **REST/HTTP** per le richieste e **SMTP** per le notifiche via email.

Il **web server**, sviluppato con **Spring Boot**, gestisce le richieste dei client e include al suo interno il database **PostgreSQL**, che garantisce la persistenza e la gestione efficiente dei dati. Questa integrazione semplifica la gestione delle risorse e riduce la latenza nelle operazioni di lettura e scrittura.

1.6.1 Gestione delle comunicazioni

Il sistema utilizza il protocollo **REST/HTTP** per lo scambio di dati in formato **JSON**, assicurando l'interoperabilità tra diverse piattaforme. Per l'invio delle notifiche agli utenti, viene impiegato il protocollo **SMTP**.

1.6.2 Caratteristiche dell'architettura

- **Scalabilità:** possibilità di scalare l'intero sistema o le singole funzionalità in base alle necessità.
- **Modularità:** sviluppo e manutenzione semplificati grazie all'organizzazione delle funzionalità in moduli indipendenti.
- **Affidabilità:** eventuali malfunzionamenti in una funzionalità non compromettono l'intero sistema.
- **Manutenibilità:** possibilità di aggiornare o sostituire singoli moduli senza interrompere l'operatività complessiva.

1.7 Strumenti Utilizzati

Lo sviluppo del progetto è supportato da una serie di strumenti software che facilitano la progettazione, lo sviluppo, il testing e la gestione del codice. Di seguito, una panoramica dettagliata dei tool impiegati:

- **Visual Studio Code:** IDE (Integrated Development Environment) scelto per lo sviluppo del progetto. VS Code offre un ambiente di sviluppo leggero ma potente, con un ampio supporto per il linguaggio Java e numerose estensioni utili. Tra i plugin utilizzati troviamo:
 - **Spring Boot Extension Pack:** per il supporto avanzato nello sviluppo di applicazioni Spring Boot.
 - **Java Extension Pack:** per il supporto completo al linguaggio Java, con funzionalità di autocompletamento, refactoring e debugging.
- **Spring Boot:** Framework Java basato su Spring, progettato per lo sviluppo di applicazioni web scalabili e strutturate secondo l'architettura a microservizi. Grazie alla sua configurazione automatica e al supporto integrato per REST API, Spring Boot semplifica la gestione del backend e garantisce un'elevata manutenibilità del codice.
- **PostgreSQL:** Sistema di gestione di database relazionale (RDBMS) scelto per la sua affidabilità, scalabilità e aderenza agli standard SQL. PostgreSQL supporta transazioni ACID (Atomicità, Coerenza, Isolamento, Durabilità) ed è ottimizzato per operazioni complesse e interrogazioni avanzate. Il database è stato configurato per garantire prestazioni elevate e sicurezza dei dati.
- **Postman:** Strumento essenziale per il testing delle API REST sviluppate con Spring Boot. Consente di effettuare richieste HTTP, validare le risposte del server e automatizzare test, facilitando così il debug e l'integrazione tra i diversi componenti del sistema.
- **Grok AI:** Tecnologia avanzata per la generazione automatica di immagini basata su intelligenza artificiale. Grok AI viene utilizzato per creare rappresentazioni visive intuitive e schematiche di concetti complessi, supportando la documentazione e la comunicazione grafica del progetto.

- **JUnit 4:** Framework per il testing unitario in Java, impiegato per validare il corretto funzionamento delle classi e dei metodi sviluppati. L'uso di test automatizzati consente di rilevare tempestivamente eventuali errori e di garantire la robustezza del codice.
- **JGraphT:** Libreria Java dedicata alla modellazione e alla gestione di strutture dati basate su grafi. Utilizzata per la rappresentazione e la manipolazione di relazioni complesse all'interno del sistema.
- **CodeMR:** Strumento avanzato per l'analisi della qualità del codice Java e la visualizzazione delle metriche software. CodeMR consente di valutare la complessità del codice, individuare problemi di design e migliorare la manutenibilità del progetto.
- **GitHub:** Piattaforma per il versionamento del codice basata su Git, utilizzata per il controllo delle versioni e la collaborazione tra gli sviluppatori. Grazie a GitHub, è possibile tracciare le modifiche al codice, gestire le revisioni e garantire un workflow ordinato ed efficiente.
- **GitHub Desktop:** Applicazione con interfaccia grafica che semplifica l'interazione con il repository GitHub direttamente dal PC. Permette di eseguire operazioni di commit, push e pull senza dover utilizzare la riga di comando, facilitando la gestione del codice per gli sviluppatori.
- **StarUML:** Software utilizzato per la modellazione di diagrammi UML (Unified Modeling Language), fondamentale nella fase di progettazione dell'architettura del sistema. StarUML consente di rappresentare visivamente classi, casi d'uso e flussi operativi.
- **diagrams.net:** Applicazione web per la creazione di diagrammi e schemi con notazione libera. Utilizzata per rappresentare graficamente flussi di dati, processi e architetture software, facilitando la comprensione e la condivisione delle informazioni progettuali.
- **WhatsApp:** Applicazione di messaggistica utilizzata come strumento di comunicazione interna tra i membri del team. Attraverso WhatsApp, è possibile coordinare le attività di sviluppo, discutere problemi tecnici e organizzare riunioni in tempo reale, garantendo un flusso comunicativo rapido ed efficiente.

- **Telegram:** Applicazione di messaggistica utilizzata dal team per discussioni tecniche e condivisione rapida di documenti, codice e aggiornamenti di progetto. Grazie ai suoi bot e alle funzionalità avanzate di gestione dei gruppi, Telegram rappresenta uno strumento utile per il coordinamento del lavoro.

2 Iterazione 1

2.1 Casi d'Uso Implementati

In questa iterazione sono stati sviluppati i seguenti casi d'uso ritenuti prioritari per lo sviluppo di **Spendly**.

ID	Titolo
UC1	Login
UC2	Registrazione
UC3	Logout
UC7	Crea gruppo
UC8	Invita memebri
UC9	Elimina membri
UC10	Modica memebri
UC11	Elimina gruppo
UC12	Accedi gruppo

Tabella 4: Iterazione1

2.1.1 UC1: Login

Attori: Utente, Sistema.

Descrizione: L'utente può autenticarsi nel sistema per accedere al proprio account.

Flusso degli eventi:

1. L'utente accede alla pagina di login.
2. Inserisce email e password.
3. Il sistema verifica le credenziali e autentica l'utente.
4. L'utente viene reindirizzato alla dashboard.

2.1.2 UC2: Registrazione

Attori: Utente, Sistema.

Descrizione: Un nuovo utente può registrarsi creando un account.

Flusso degli eventi:

1. L'utente accede alla pagina di registrazione.
2. Inserisce nome, email e password.
3. Il sistema verifica che l'email non sia già registrata.
4. Se la verifica è superata, il sistema crea l'account e lo memorizza.
5. L'utente viene reindirizzato alla dashboard.

2.1.3 UC3: Logout

Attori: Utente, Sistema.

Descrizione: L'utente può terminare la sessione ed effettuare il logout.

Flusso degli eventi:

1. L'utente clicca su "Logout".
2. Il sistema invalida la sessione e mostra la schermata di login.

2.1.4 UC7: Creazione Gruppo

Attori: Utente amministratore, Sistema.

Descrizione: L'utente può creare un nuovo gruppo di spese per la condivisione con altri membri.

Flusso degli eventi:

1. L'utente clicca su "Crea Gruppo".
2. Inserisce il nome del gruppo e una descrizione opzionale.
3. Il sistema crea il gruppo e assegna l'utente come amministratore.
4. L'utente viene reindirizzato alla pagina del gruppo.

2.1.5 UC8: Invita Membri

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può invitare altri utenti a unirsi al gruppo di spese.

Flusso degli eventi:

1. L'amministratore accede alla pagina del gruppo.
2. Clicca su "Invita Membri" e inserisce l'email degli utenti da invitare.
3. Il sistema invia un'email con l'invito e memorizza la richiesta.
4. Gli utenti ricevono l'invito e possono accettarlo per entrare nel gruppo.

2.1.6 UC9: Elimina Membri

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può rimuovere un membro dal gruppo.

Flusso degli eventi:

1. L'amministratore accede alla lista dei membri del gruppo.
2. Seleziona il membro da rimuovere e clicca su "Elimina".
3. Il sistema rimuove il membro e aggiorna la lista.

2.1.7 UC10: Modifica Membri

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può modificare i dettagli dei membri (ad esempio assegnare nuovi ruoli).

Flusso degli eventi:

1. L'amministratore accede alla lista dei membri.

2. Seleziona un membro e modifica i dettagli (es. ruolo nel gruppo).
3. Il sistema aggiorna i dati e notifica il cambiamento.

2.1.8 UC11: Eliminazione Gruppo

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può eliminare definitivamente un gruppo di spese.

Flusso degli eventi:

1. L'amministratore accede alle impostazioni del gruppo.
2. Clicca su "Elimina Gruppo".
3. Il sistema chiede conferma prima di procedere.
4. Se confermato, il gruppo e tutte le sue spese vengono eliminate.

2.1.9 UC12: Accesso a un Gruppo

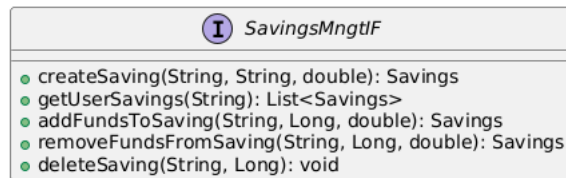
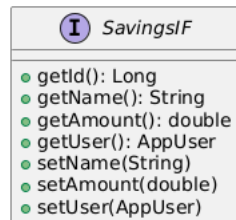
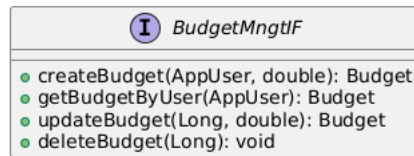
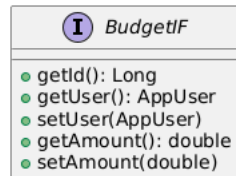
Attori: Utente, Sistema.

Descrizione: Un utente può accedere a un gruppo di spese a cui è stato invitato.

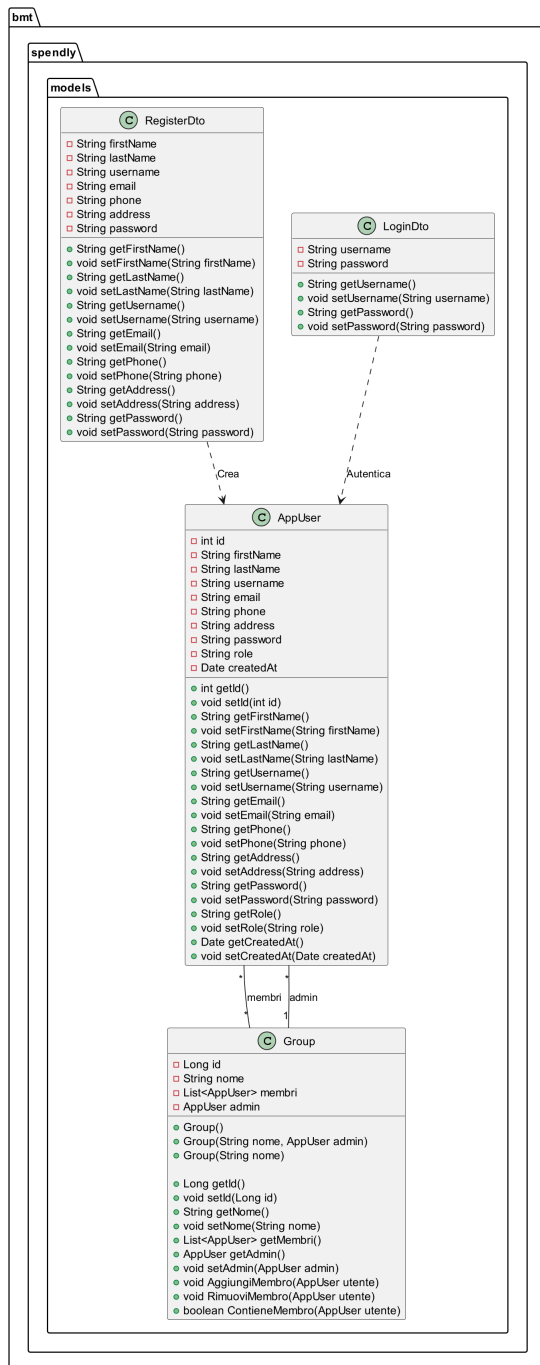
Flusso degli eventi:

1. L'utente riceve un invito via email o notifica nell'app.
2. Clicca sul link di invito e accede alla web-app.
3. Il sistema verifica la validità dell'invito e aggiunge l'utente al gruppo.
4. L'utente viene reindirizzato alla pagina del gruppo.

2.2 Diagramma delle Interfacce per Budget e Risparmio



2.3 UML Class Diagram per tipi di dato



LoginDto → AppUser:

- Rappresenta il processo di autenticazione nel sistema.
- La relazione mostra che la classe **LoginDto** contiene i dati necessari (username e password) per verificare l'identità di un utente già registrato (**AppUser**).
- In altre parole, **LoginDto** fornisce un mezzo per confermare che un utente (**AppUser**) ha accesso al sistema.

RegisterDto → AppUser:

- Rappresenta il processo di registrazione di un nuovo utente.
- La relazione indica che la classe **RegisterDto** contiene i dati necessari (nome, cognome, email, password, ecc.) per creare un nuovo account utente (**AppUser**) all'interno del sistema.
- Questo significa che la classe **RegisterDto** viene utilizzata per tradurre i dati di registrazione in un'istanza valida della classe **AppUser**.

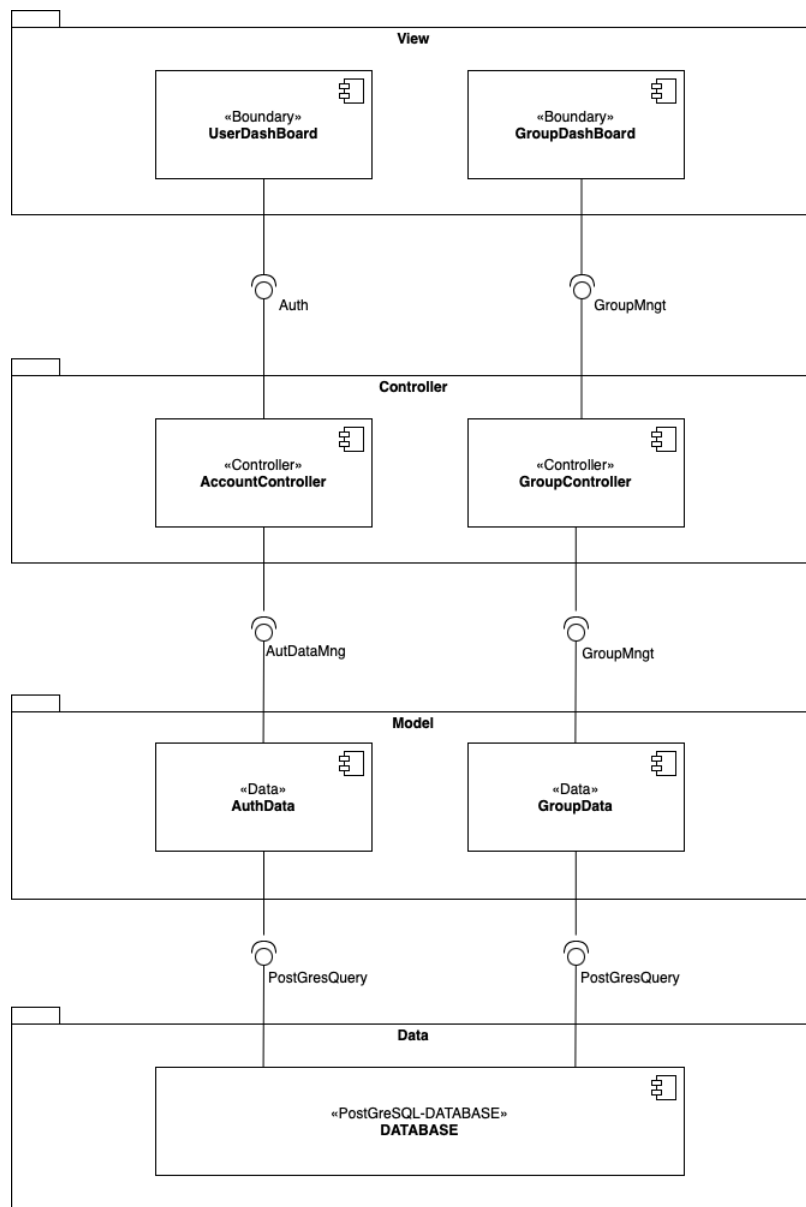
Group → AppUser (admin):

- Rappresenta il ruolo di amministratore per un gruppo.
- Ogni gruppo (**Group**) ha un amministratore specifico (**AppUser**) che è responsabile della gestione del gruppo.
- La relazione è uno-a-uno (1:1), il che significa che ogni gruppo ha un solo amministratore, ma un amministratore può gestire più gruppi.
- Questo legame indica che l'amministratore è una figura centrale per la gestione delle attività e dei membri del gruppo.

AppUser ↔ Group (membri):

- Rappresenta i membri di un gruppo e la loro relazione con i gruppi.
- La relazione è multi-a-molti ($m : n$), il che significa che:
 - Ogni utente (**AppUser**) può essere membro di più gruppi (**Group**).
 - Allo stesso tempo, ogni gruppo può avere più utenti come membri.
- Questo legame mostra che l'utente e il gruppo sono fortemente interconnessi, poiché i gruppi esistono per aggregare utenti, e gli utenti possono partecipare a più attività o comunità rappresentate dai gruppi.

2.4 UML Component Diagram



Partendo dai casi d'uso selezionati per questa iterazione e procedendo con l'utilizzo delle euristiche di design, è stato possibile progettare l'architettura software del sistema **Spendly**. I componenti sono organizzati secondo il pattern **MVC (Model-View-Controller)**, con la suddivisione in:

- **Boundary** - Interfaccia utente, responsabile dell'interazione con l'utente finale.
- **Controller** - Gestione logica di business.
- **Model** - Gestione dei dati e accesso al database.
- **Service** - Servizi di sicurezza e autenticazione.
- **Database** - PostgreSQL.

2.5 UML Deployment Diagram

Mediante l'utilizzo di un Deployment Diagram UML è possibile mostrare la rappresentazione hardware e software del sistema. In Figura 29 vengono mostrati i componenti contenuti nei seguenti nodi:

- **Admin:** Interfaccia web utilizzata da un utente admin per creare, gestire e aggiungere membri ai gruppi. Il nodo contiene il componente *AppUserMngt* per la gestione degli utenti.
- **User:** Interfaccia web utilizzata dagli utenti per accedere alla piattaforma e gestire i gruppi. Contiene i componenti front-end *AppUserMngt* per la gestione dell'account utente e *GroupMngt* per la gestione dei gruppi.
- **Web Server:** Nodo server che espone le API necessarie per la gestione degli utenti e dei gruppi. Contiene i seguenti componenti:
 - *AppUserController*: Controller per la gestione delle operazioni relative agli utenti.
 - *GroupController*: Controller per la gestione delle operazioni sui gruppi.
 - *AppUser*: Modulo dati per la memorizzazione e gestione delle informazioni utente.
 - *Group*: Modulo dati per la memorizzazione e gestione delle informazioni sui gruppi.
- **Database:** PostgreSQL

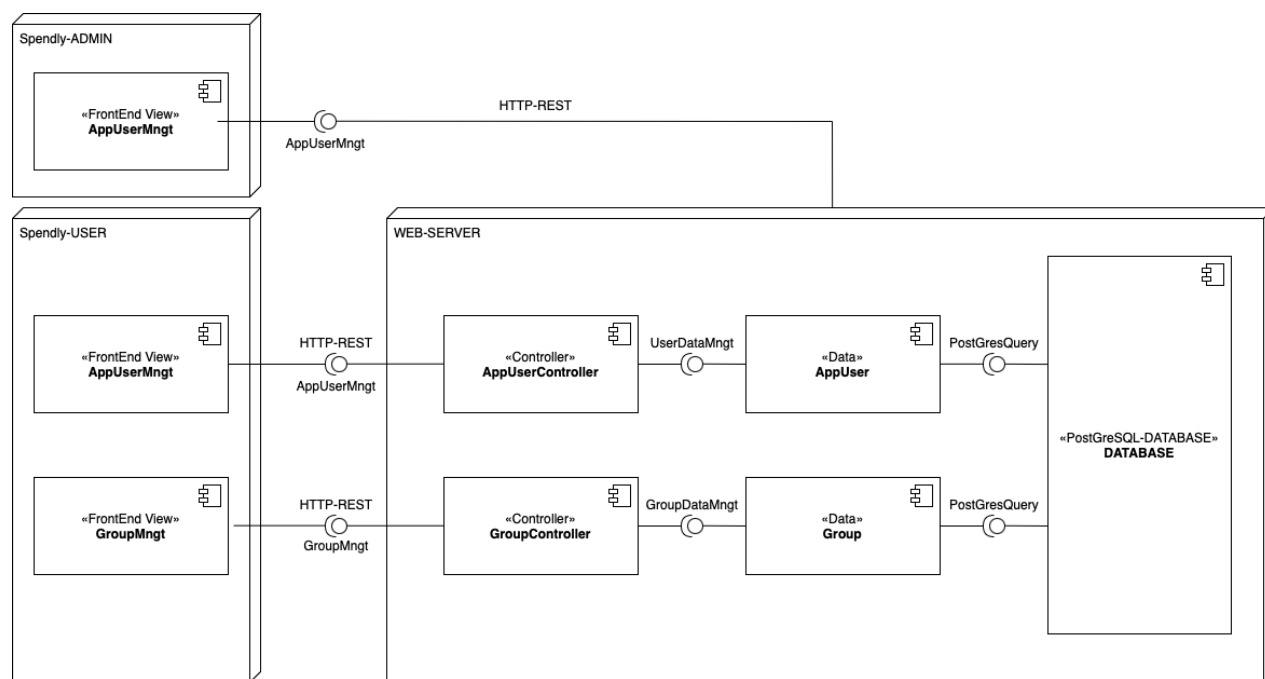


Figura 4: UML Deployment Diagram

2.6 Testing

2.6.1 Analisi Statica - CodMR

L'analisi statica del codice è stata gestita tramite il tool CodeMR.

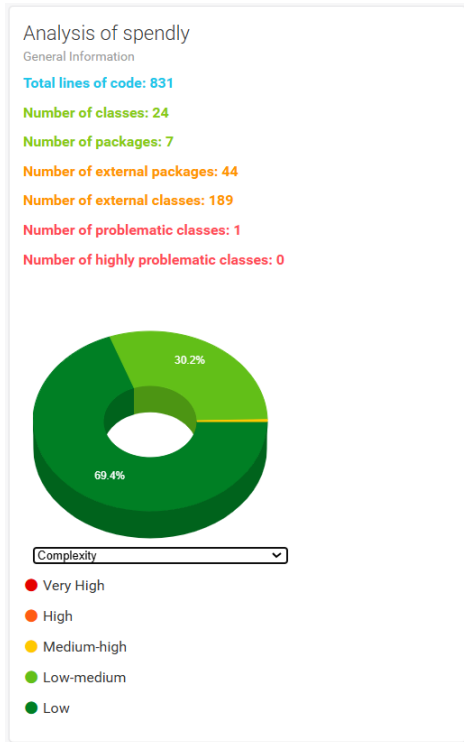


Figura 5: Complexity

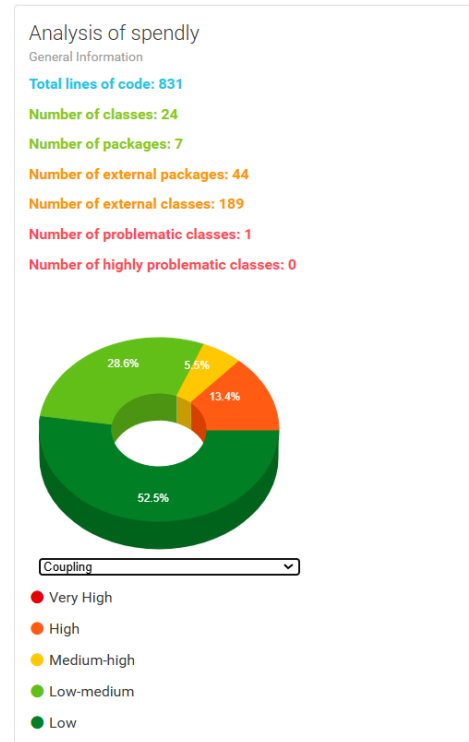


Figura 6: Coupling

Classes with high coupling, high complexity, low cohesion (#0)											
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	WMC	RFC	CBO	LCOM	
Classes with high coupling, high complexity (#0)											
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	WMC	RFC	NOM	
Classes with high coupling (#1)											
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC	
1	AccountController	■	■	■	■	111	21	4	17	59	
Classes with high complexity (#0)											
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	WMC	RFC	NOM	NOF	DIT

Figura 7: Problemi classi

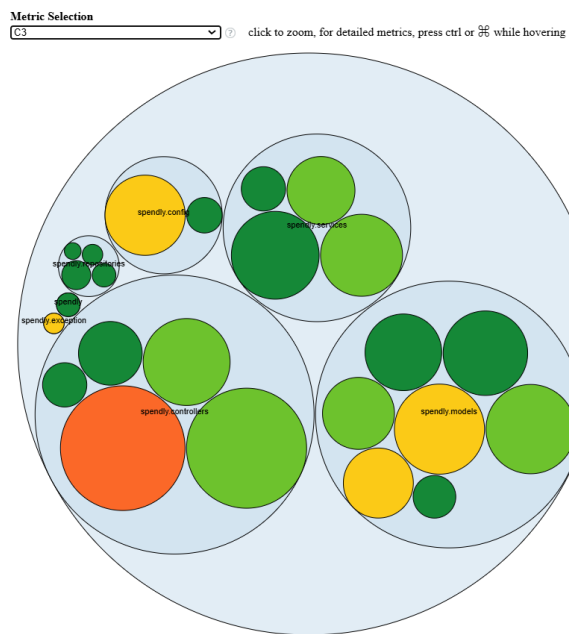


Figura 8: Struttura dei package

2.6.2 Analisi Dinamica - JUnit

L'analisi dinamica del codice è stata condotta utilizzando JUnit per l'esecuzione di test automatizzati sui metodi delle principali classi, e Postman per

verificare il corretto funzionamento delle API implementate nei controller. In particolare, con JUnit sono stati sviluppati test specifici per validare i metodi delle classi relative alla gestione dei gruppi, quali Group, GroupService e GroupController.

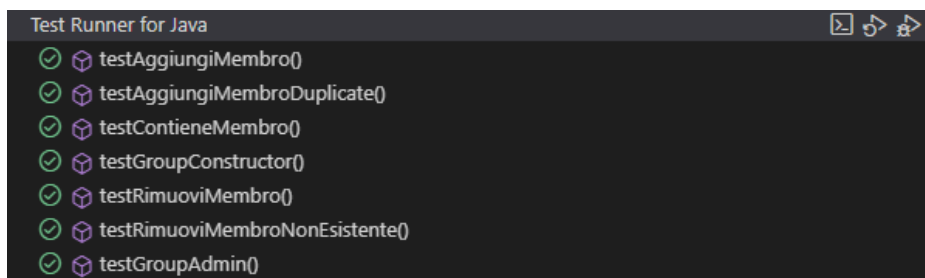


Figura 9: Test per la classe Group

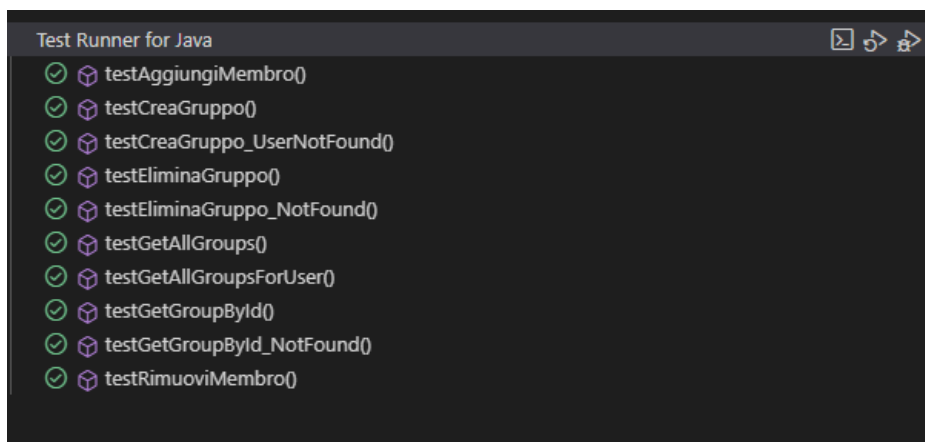


Figura 10: Test per la classe GroupService

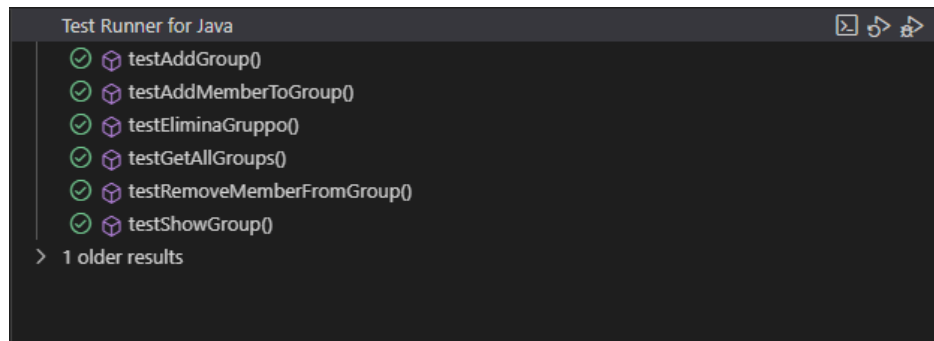


Figura 11: Test per la classe GroupController

2.6.3 API Esposte

Questa sezione documenta le API principali del sistema **Spendly**, includendo autenticazione, gestione dei gruppi e visualizzazione. Ogni test verrà mostrato con un'immagine dei risultati.

Registrazione Utente

- **Endpoint:** POST /account/register
- **Descrizione:** Consente a un nuovo utente di registrarsi al sistema.
- **Parametri:**
 - nome (string) - Nome utente.
 - cognome (string) - Nome utente.
 - username (string) - Username utente(non accetta duplicati).
 - email (string) - Email dell'utente(non accetta duplicati).
 - telefono (string) - Telefono utente.
 - indirizzo (string) - Indirizzo utente.
 - password (string) - Password scelta dall'utente.
- **Risultato:**



Figura 12: Risultato API Registrazione

Login Utente

- **Endpoint:** POST /account/login
- **Descrizione:** Permette a un utente registrato di accedere al sistema.
- **Parametri:**
 - username (string) - Username dell'utente.
 - password (string) - Password dell'utente.
- **Risultato:**

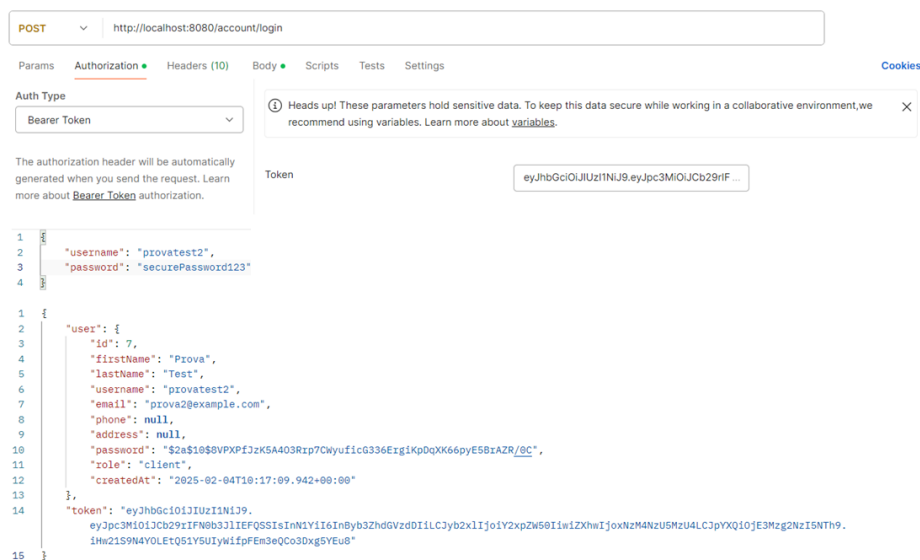


Figura 13: Risultato API Login

Creazione di un Gruppo

- **Endpoint:** POST /api/groups
- **Descrizione:** Permette la creazione di un nuovo gruppo da parte dell'utente.
- **Parametri:**
 - name (string) - Nome del gruppo.
 - username (string) - Username dell'utente che crea il gruppo.
- **Risultato:**

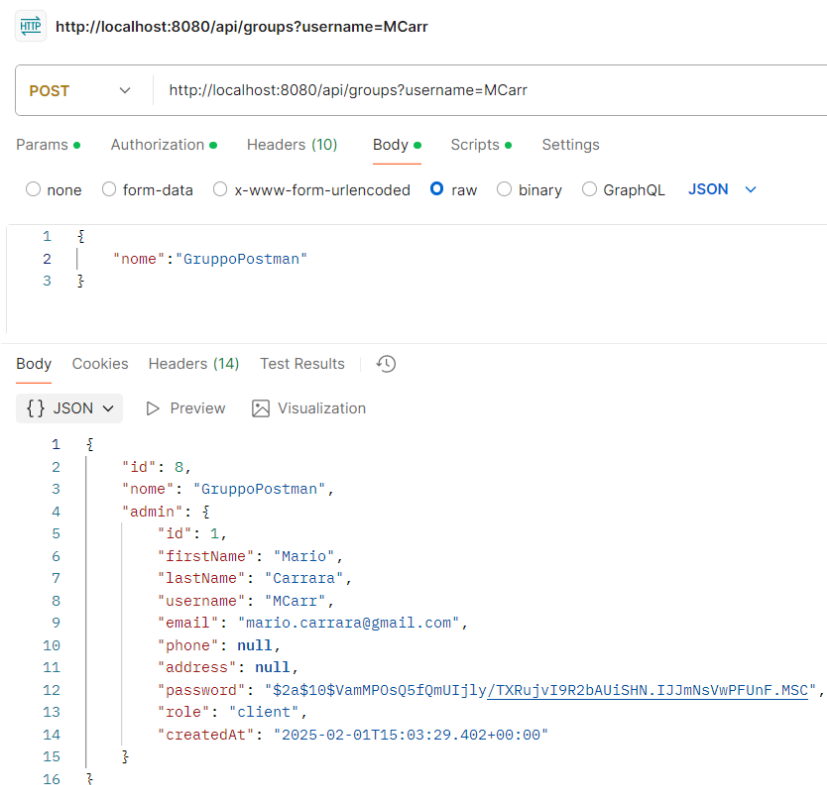


Figura 14: Risultato API Creazione Gruppo

Inserimento di un utente in un gruppo

- **Endpoint:** POST /api/groups/group_id/members
- **Descrizione:** Permette all'amministratore di inserire un utente in un gruppo.
- **Parametri:**
 - adminUsername (string) - Username dell'admin del gruppo.
 - memberUsername (string) - Username dell'utente da inserire.
- **Risultato:**

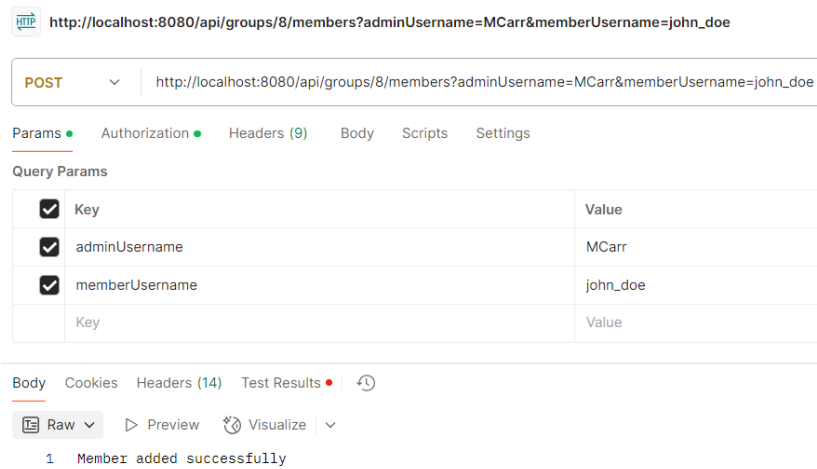


Figura 15: Risultato API Inserimento utente

Visualizzazione dei Gruppi

- **Endpoint:** GET `/api/groups`
- **Descrizione:** Restituisce la lista di tutti i gruppi a cui l'utente appartiene.
- **Parametri:**
 - `username` (string) - Username dell'utente.
- **Risultato:**

HTTP http://localhost:8080/api/groups?username=john_doe

GET http://localhost:8080/api/groups?username=john_doe

Params • Authorization • Headers (8) Body Scripts Settings

Query Params

Key	Value
username	john_doe
Key	Value

Body Cookies Headers (14) Test Results ↺

{ } JSON ▾ ▷ Preview 🔍 Visualize ▾

```

1  [
2    {
3      "id": 8,
4      "nome": "GruppoPostman",
5      "admin": {
6        "id": 1,
7        "firstName": "Mario",
8        "lastName": "Carrara",
9        "username": "MCarr",
10       "email": "mario.carrara@gmail.com",
11       "phone": null,
12       "address": null,
13       "password": "$2a$10$VamMP0sQ5fQmUIjly/TXRujvI9R2bAUiSHN.IJJmNsVwPFUnF.MSC",
14       "role": "client",
15       "createdAt": "2025-02-01T15:03:29.402+00:00"
      }
    }
  ]

```

Figura 16: Risultato API Visualizzazione Gruppi

3 Iterazione 2

3.1 Casi d'Uso Implementati

In questa iterazione sono stati sviluppati i seguenti casi d'uso ritenuti prioritari per lo sviluppo di **Spendly**.

ID	Titolo
UC4	Crea alert di gruppo
UC6	Elimina alert
UC12	Accedi gruppo
UC13	Inserisci spesa
UC14	Elimina spesa
UC15	Modifica spesa
UC16	Visualizza spese
UC17	Ricalcolo debiti

Tabella 5: Iterazione2

3.1.1 UC4: Creazione Alert di Gruppo

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può creare un alert per segnalare il raggiungimento di una soglia limite di spesa.

Flusso degli eventi:

1. L'amministratore accede alla pagina del gruppo.
2. Clicca su "Crea Alert".
3. Inserisce il limite di spesa e una descrizione opzionale.
4. Il sistema salva l'alert e lo associa al gruppo.

3.1.2 UC6: Eliminazione Alert

Attori: Utente amministratore, Sistema.

Descrizione: L'amministratore di un gruppo può eliminare un alert impostato in precedenza.

Flusso degli eventi:

1. L'amministratore accede alla lista degli alert del gruppo.
2. Seleziona un alert e clicca su "Elimina".
3. Il sistema chiede conferma e poi rimuove l'alert.

3.1.3 UC13: Inserimento Spesa

Attori: Utente, Sistema.

Descrizione: Un utente appartenente a un gruppo può aggiungere una spesa condivisa.

Flusso degli eventi:

1. L'utente accede alla pagina del gruppo.
2. Clicca su "Aggiungi Spesa".
3. Inserisce l'importo, la descrizione e seleziona i partecipanti.
4. Il sistema registra la spesa e aggiorna il bilancio del gruppo.

3.1.4 UC14: Eliminazione Spesa

Attori: Utente, Sistema.

Descrizione: Un utente può eliminare una spesa precedentemente inserita.

Flusso degli eventi:

1. L'utente accede alla lista delle spese del gruppo.
2. Seleziona una spesa e clicca su "Elimina".
3. Il sistema chiede conferma e poi rimuove la spesa dal gruppo.

3.1.5 UC15: Modifica Spesa

Attori: Utente, Sistema.

Descrizione: Un utente può modificare i dettagli di una spesa condivisa nel gruppo.

Flusso degli eventi:

1. L'utente accede alla lista delle spese.
2. Seleziona una spesa e clicca su "Modifica".
3. Modifica i dati della spesa (importo, descrizione, partecipanti).
4. Il sistema aggiorna la spesa e ricalcola i bilanci.

3.1.6 UC16: Visualizzazione Spese

Attori: Utente, Sistema.

Descrizione: Un utente può visualizzare tutte le spese del gruppo di cui fa parte.

Flusso degli eventi:

1. L'utente accede alla pagina del gruppo.
2. Clicca su "Visualizza Spese".
3. Il sistema mostra l'elenco delle spese con dettagli su importo, data e partecipanti.

3.1.7 UC17: Ricalcolo Debiti

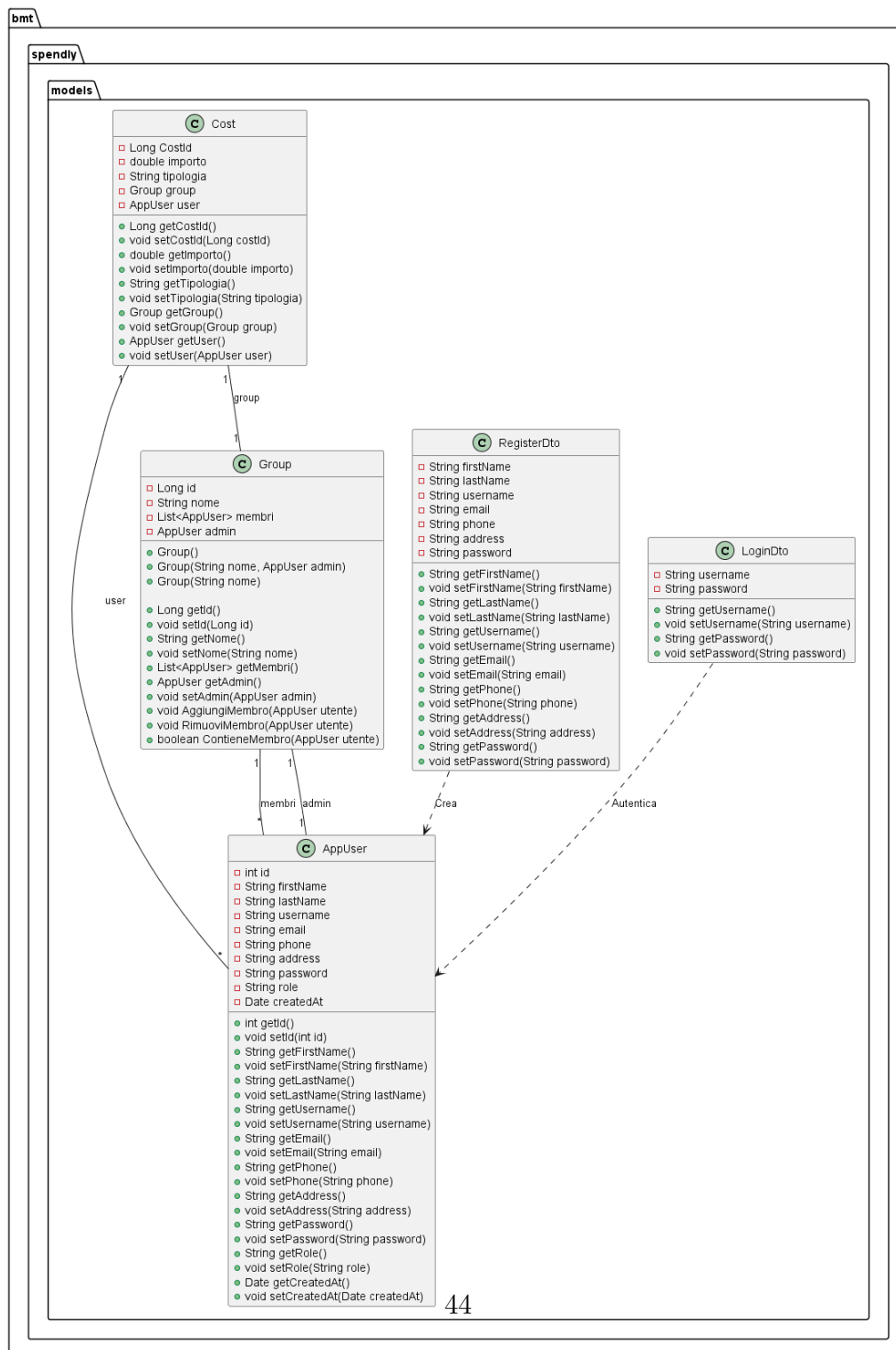
Attori: Utente, Sistema.

Descrizione: Un utente può calcolare il riepilogo dei debiti tra i membri del gruppo.

Flusso degli eventi:

1. L'utente accede alla pagina del gruppo.
2. Clicca su "Calcola Debiti".
3. Il sistema analizza le spese e genera un riepilogo dei debiti tra i membri.
4. L'utente visualizza il riepilogo e le modalità di saldo consigliate.

3.2 UML Class Diagram per tipi di dato



LoginDto → AppUser:

- Rappresenta il processo di autenticazione nel sistema.
- La relazione mostra che la classe **LoginDto** contiene i dati necessari (username e password) per verificare l'identità di un utente già registrato (**AppUser**).
- In altre parole, **LoginDto** fornisce un mezzo per confermare che un utente (**AppUser**) ha accesso al sistema.

RegisterDto → AppUser:

- Rappresenta il processo di registrazione di un nuovo utente.
- La relazione indica che la classe **RegisterDto** contiene i dati necessari (nome, cognome, email, password, ecc.) per creare un nuovo account utente (**AppUser**) all'interno del sistema.
- Questo significa che la classe **RegisterDto** viene utilizzata per tradurre i dati di registrazione in un'istanza valida della classe **AppUser**.

Group → AppUser (admin):

- Rappresenta il ruolo di amministratore per un gruppo.
- Ogni gruppo (**Group**) ha un amministratore specifico (**AppUser**) che è responsabile della gestione del gruppo.
- La relazione è uno-a-uno (1:1), il che significa che ogni gruppo ha un solo amministratore, ma un amministratore può gestire più gruppi.
- Questo legame indica che l'amministratore è una figura centrale per la gestione delle attività e dei membri del gruppo.

AppUser ↔ Group (membri):

- Rappresenta i membri di un gruppo e la loro relazione con i gruppi.
- La relazione è multi-a-molti ($m : n$), il che significa che:
 - Ogni utente (**AppUser**) può essere membro di più gruppi (**Group**).
 - Allo stesso tempo, ogni gruppo può avere più utenti come membri.

- Questo legame mostra che l'utente e il gruppo sono fortemente interconnessi, poiché i gruppi esistono per aggregare utenti, e gli utenti possono partecipare a più attività o comunità rappresentate dai gruppi.

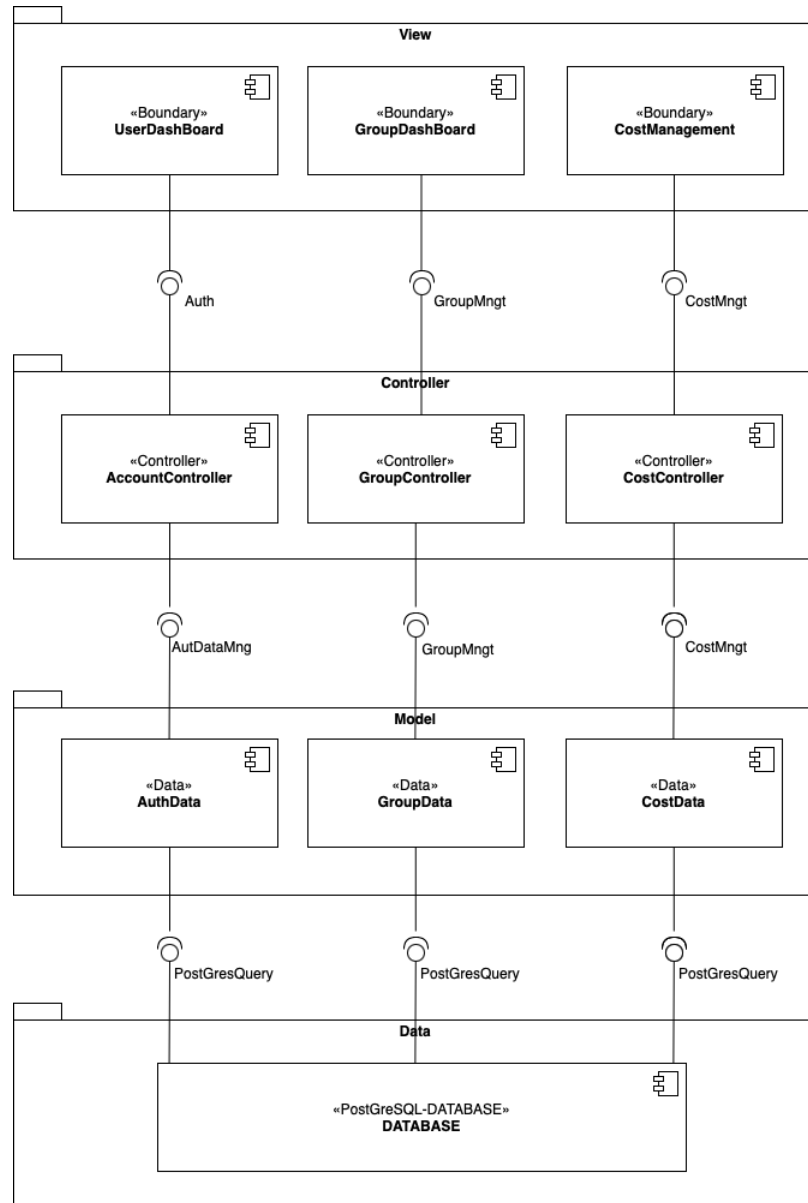
Cost → Group:

- Indica che un costo (**Cost**) può essere associato a un gruppo (**Group**).
- La relazione è multi-a-uno (1:n), il che significa che:
 - Un costo appartiene a un singolo gruppo.
 - Tuttavia, un gruppo può avere più costi associati.
- Questa relazione consente di rappresentare costi condivisi o transazioni relative a un gruppo specifico.

Cost → AppUser:

- Indica che un costo (**Cost**) è associato a un utente (**AppUser**).
- La relazione è multi-a-uno (1:n), il che significa che:
 - Un costo è creato o sostenuto da un singolo utente.
 - Tuttavia, un utente può avere più costi associati.
- Questa relazione mostra che ogni costo ha un responsabile (l'utente) che lo ha sostenuto.

3.3 UML Component Diagram



In questa iterazione, aggiorniamo il diagramma dei componenti rispetto all'iterazione 1 data l'introduzione dei costi. Partendo dai casi d'uso selezionati per questa iterazione e procedendo con l'utilizzo delle euristiche

di design, è stato possibile progettare l'architettura software del sistema Spendly.

3.4 UML Deployment Diagram

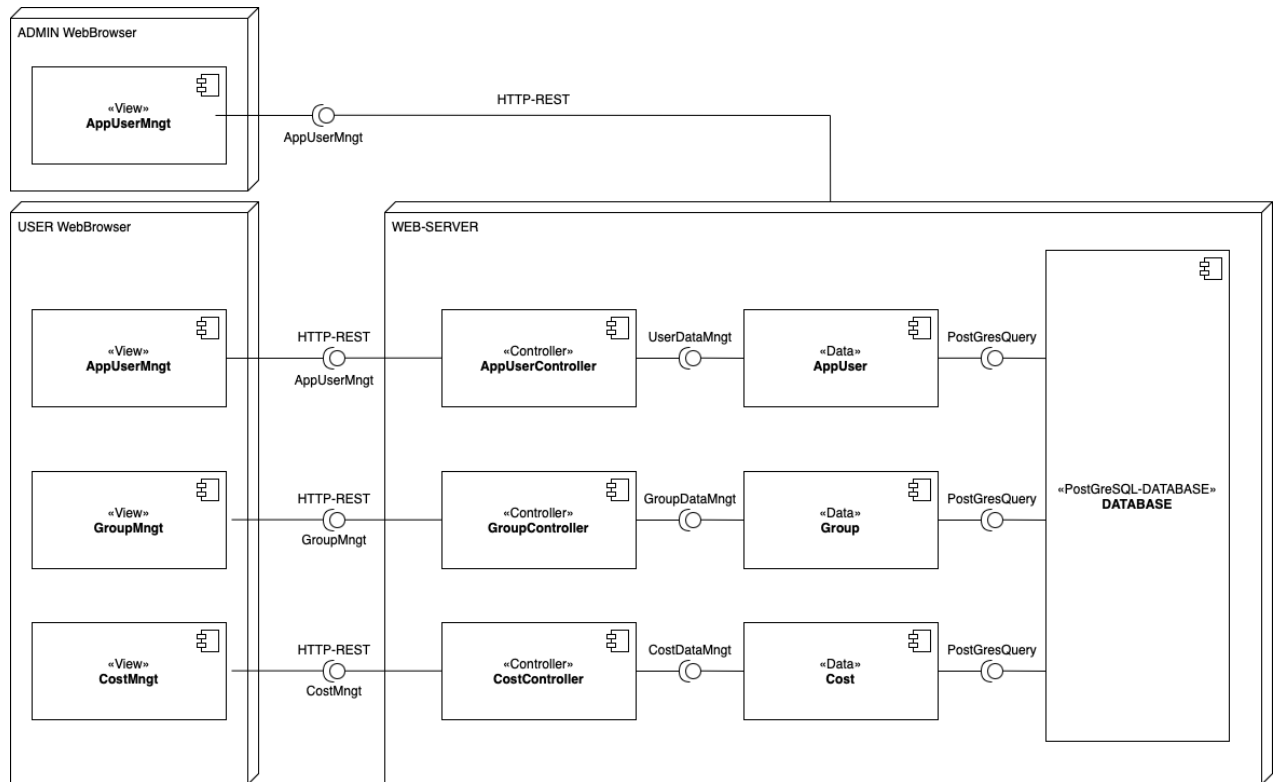


Figura 17: UML Deployment Diagram

3.5 Testing

3.5.1 Analisi Statika - CodeMR

Analysis of spendly

General Information

Total lines of code: 1057

Number of classes: 30

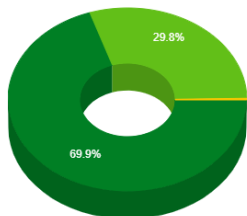
Number of packages: 7

Number of external packages: 44

Number of external classes: 216

Number of problematic classes: 1

Number of highly problematic classes: 0



Complexity

- Very High
- High
- Medium-high
- Low-medium
- Low

Figura 18: Complexity

Analysis of spendly

General Information

Total lines of code: 1057

Number of classes: 30

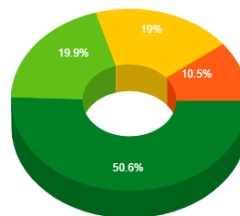
Number of packages: 7

Number of external packages: 44

Number of external classes: 216

Number of problematic classes: 1

Number of highly problematic classes: 0



Coupling

- Very High
- High
- Medium-high
- Low-medium
- Low

Figura 19: Coupling

Classes with high coupling, high complexity, low cohesion (#0)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	WMC	RFC	CBO	LCAM
Classes with high coupling, high complexity (#0)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	WMC	RFC	NOM
Classes with high coupling (#1)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	AccountController	■	■	■	■	111	21	4	17	59
Classes with high complexity (#0)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	WMC	RFC	NOM	NOF

Figura 20: Problemi classi

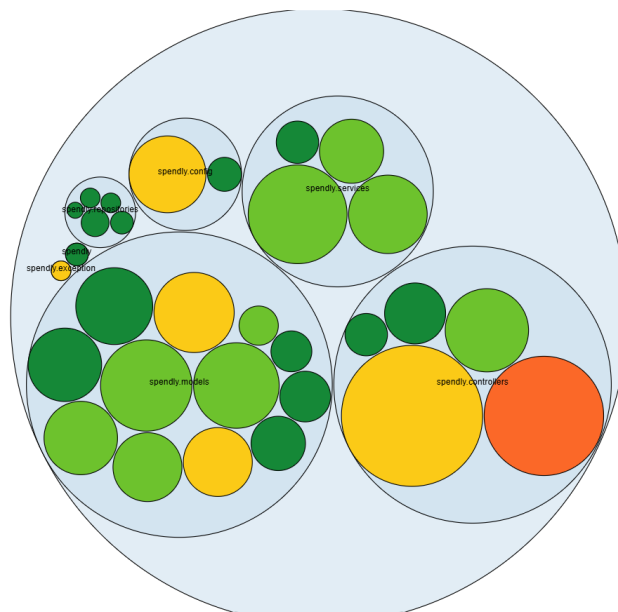


Figura 21: Struttura dei package

3.5.2 API Costi

Questa sezione documenta le API relative alla gestione dei **costi** nel sistema **Spendly**, inclusi la creazione, l'eliminazione e la visualizzazione dei costi di utenti e gruppi. Ogni test verrà mostrato con un'immagine dei risultati.

Aggiunta di un Costo

- **Endpoint:** POST /api/costs?username={username}
- **Descrizione:** Permette all'utente autenticato di aggiungere un nuovo costo, eventualmente associandolo a un gruppo.
- **Parametri:**
 - importo (double) - Importo della spesa.
 - tipologia (string) - Tipo di spesa.
 - groupId (integer, opzionale) - ID del gruppo a cui associare il costo.

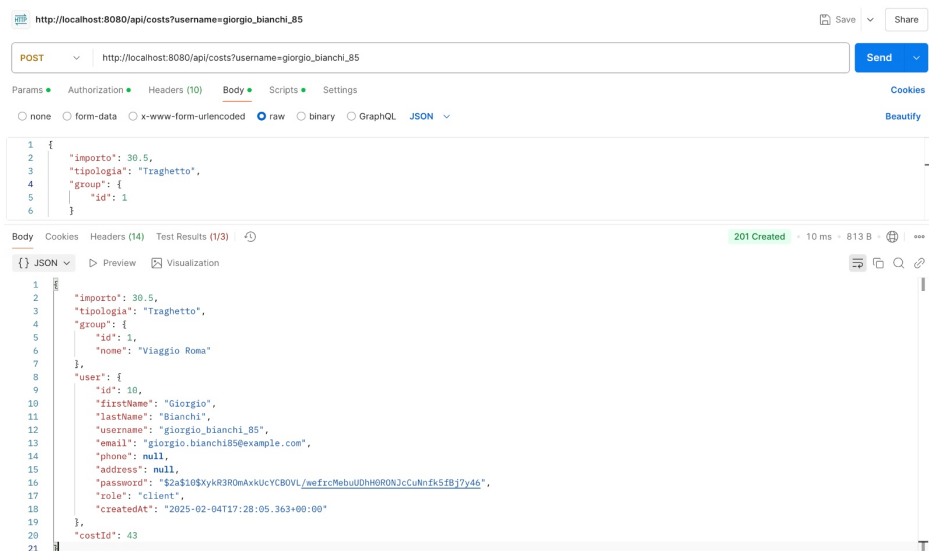


Figura 22: Risultato API Aggiunta Costo

Eliminazione di un Costo

- **Endpoint:** DELETE /api/costs/{costId}
- **Descrizione:** Permette all'utente autenticato di eliminare un costo precedentemente registrato.
- **Parametri:**
 - {costId} (integer) - ID del costo da eliminare.

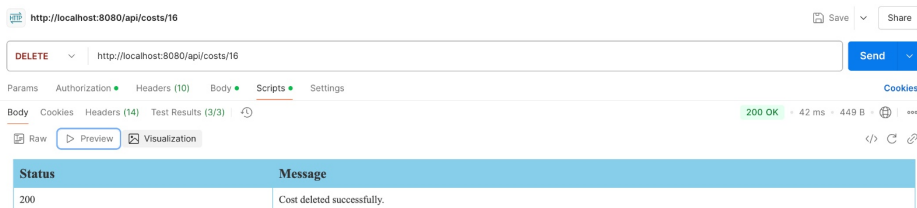


Figura 23: Risultato API Eliminazione Costo

Visualizzazione Costi Utente

- **Endpoint:** GET /api/costs?username={username}
- **Descrizione:** Restituisce la lista di tutti i costi registrati dall'utente autenticato.

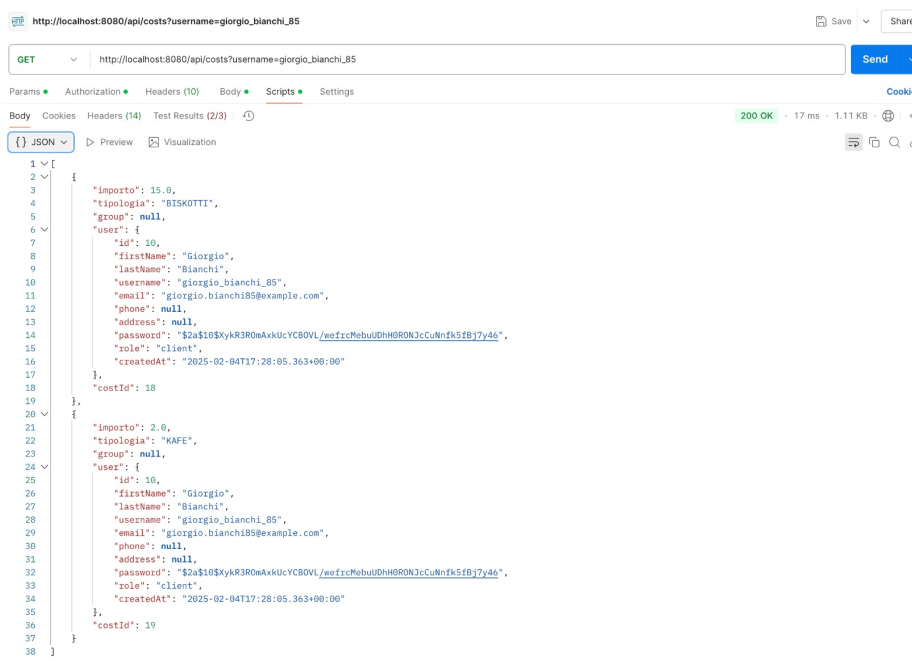
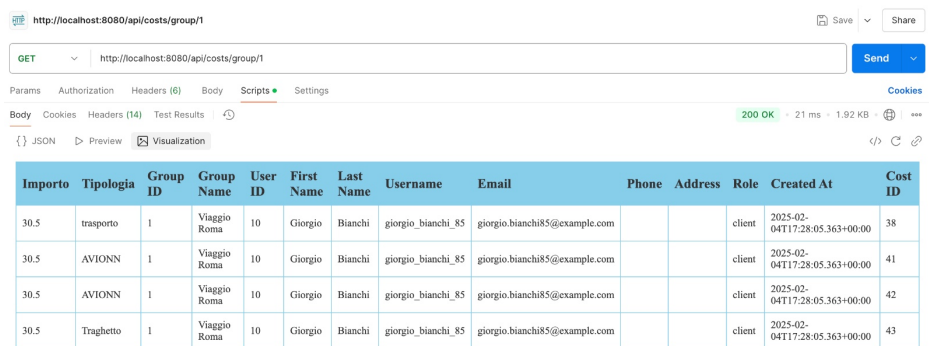


Figura 24: Risultato API Visualizzazione Costi Utente

Visualizzazione Costi di un Gruppo

- **Endpoint:** GET /api/costs/group/{groupId}
- **Descrizione:** Restituisce la lista di tutti i costi associati a un gruppo, accessibile solo dai membri del gruppo.



Importo	Tipologia	Group ID	Group Name	User ID	First Name	Last Name	Username	Email	Phone	Address	Role	Created At	Cost ID
30.5	trasporto	1	Viaggio Roma	10	Giorgio	Bianchi	giorgio_bianchi_85	giorgio.bianchi85@example.com			client	2025-02-04T17:28:05.363+00:00	38
30.5	AVIONN	1	Viaggio Roma	10	Giorgio	Bianchi	giorgio_bianchi_85	giorgio.bianchi85@example.com			client	2025-02-04T17:28:05.363+00:00	41
30.5	AVIONN	1	Viaggio Roma	10	Giorgio	Bianchi	giorgio_bianchi_85	giorgio.bianchi85@example.com			client	2025-02-04T17:28:05.363+00:00	42
30.5	Traghetto	1	Viaggio Roma	10	Giorgio	Bianchi	giorgio_bianchi_85	giorgio.bianchi85@example.com			client	2025-02-04T17:28:05.363+00:00	43

Figura 25: Risultato API Visualizzazione Costi Gruppo

3.6 Algoritmo ottimizza debiti

3.6.1 PseudoCodice

```

creaDizionario(ListaSpese):
    Dizionario credit <- Dizionario (utente, credit_score)
    inizializzo tutti i credi score a 0

    for item in ListaSpese:
        pagante = item.pagante
        beneficiari[] = item.beneficiari
        importo = item.importo
        quota = importo / len(beneficiari)

        // Se il pagante è tra i beneficiari, deve recuperare solo la quota degli altri
        if pagante in beneficiari:
            credit_score[pagante] += importo - quota
            beneficiari.remove(pagante) // Evito di sottrargli la quota due volte
        else:
            credit_score[pagante] += importo

        // Aggiorno i crediti dei beneficiari sottraendo la loro quota
        for beneficiario in beneficiari:
            credit_score[beneficiario] -= quota

    return credit
    
```

Figura 26: Funzione CreaDizionario


```
calcolaDebiti(ListaSpese):
    Dizionario credit_score = creaDizionario(ListaSpese)
    Dizionario positivi = {} // Lista di chi deve ricevere soldi
    Dizionario negativi = {} // Lista di chi deve dare soldi

    // Separiamo i creditori dai debitori
    for each utente in k:
        if(credit_score[utente]>0) positivi.add(utente,credit_score[utente])
        if(credit_score[utente]<0) negativi.add(utente,credit_score[utente])

    Ordino i negativi in ordine crescente //( es: -50, -30, 20, -5)
    lista_transazioni = []

    // Associa creditori e debitori
    for positivo in positivi:
        credito=credit_score[positivo]
        i = 0
        while i < len(negativi) and credito > 0:
            negativo = negativi[i]
            debito_assoluto = abs(negativo.credit_score)

            if debito_assoluto <= credito:
                lista_transazioni.add((negativo.utente, positivo.utente, debito_assoluto))
                credito -= debito_assoluto
                negativi.remove(negativo) // Rimuovo il debitore se il debito è saldato
            else:
                lista_transazioni.add((negativo.utente, positivo.utente, credito))
                negativi[i].credit_score += credito
                credito = 0

            if credito == 0:
                break // Passo al prossimo creditore

        i += 1 // Avanzo al prossimo debitore solo se quello attuale è stato saldato

    return lista_transazioni
```

Figura 27: Funzione CalcolaDebiti

3.6.2 Descrizione

La funzione `CreaDizionario` prende in input una matrice di spese e costruisce un dizionario in cui:

- La chiave è lo *username* dell'utente.
- Il valore è il suo *credit score*, che indica quanto deve ricevere o restituire.

Scorrendo i record della matrice, la funzione aggiorna progressivamente i *credit score* degli utenti.

Se il *credit score* è positivo, l'utente deve ricevere soldi. Se invece è negativo, deve restituire una somma.

La funzione `calcolaDebiti` utilizza `CreaDizionario` per ottenere il saldo di ogni utente. A questo punto, suddivide gli utenti in due liste:

- **Creditori:** utenti con *credit score* positivo.
- **Debitori:** utenti con *credit score* negativo, ordinati in modo crescente.

Per ogni creditore, si scorre l'elenco dei debitori per saldare i crediti:

1. Se il debito di un utente è minore o uguale al credito disponibile:
 - Si sottrae l'intero importo dal credito.
 - La transazione viene registrata.
 - Il debitore viene rimosso dalla lista, poiché ha estinto il suo debito.
 - Se il credito arriva a zero, si interrompe l'iterazione e si passa al creditore successivo.
2. Se invece il debito è maggiore del credito:
 - Si utilizza tutto il credito disponibile per ridurre il debito.
 - La transazione viene registrata.
 - Il debitore rimane nella lista, poiché ha ancora un saldo negativo da coprire.
 - Il credito diventa zero e si passa al creditore successivo.

3.6.3 Complessità

La funzione `CreaDizionario` ha:

- Un ciclo esterno che itera su tutti gli elementi della matrice $\rightarrow O(n)$
- Un ciclo interno che itera su tutti i beneficiari di ogni record $\rightarrow O(n)$

Essendo i due cicli annidati, la complessità totale è: $O(n^2)$.

La funzione `CalcolaDebiti` esegue le seguenti operazioni:

1. Richiama `CreaDizionario` $\rightarrow O(n^2)$
2. Itera sul dizionario per separare creditori e debitori $\rightarrow O(n)$
3. Ordina la lista dei debitori (merge sort) $\rightarrow O(n \log n)$
4. Associa creditori e debitori
 - Itera su tutti i creditori $\rightarrow O(n)$
 - Per ogni creditore, nel caso peggiore, può iterare su tutti i debitori $\rightarrow O(n)$
 - Complessità totale del ciclo annidato $\rightarrow O(n^2)$

Sommiamo i contributi principali:

$$O(n^2) + O(n) + O(n \log n) + O(n^2) = O(n^2)$$

Complessità finale: $O(n^2)$

4 Iterazione 3

4.1 Introduzione

In questa iterazione, abbiamo implementato la gestione del Budget, permettendo agli utenti di monitorare e gestire i propri fondi disponibili. Inoltre, abbiamo introdotto la funzionalità di Risparmio, che consente agli utenti di creare obiettivi di risparmio per mettere da parte denaro per esigenze future.

4.2 Casi d'Uso

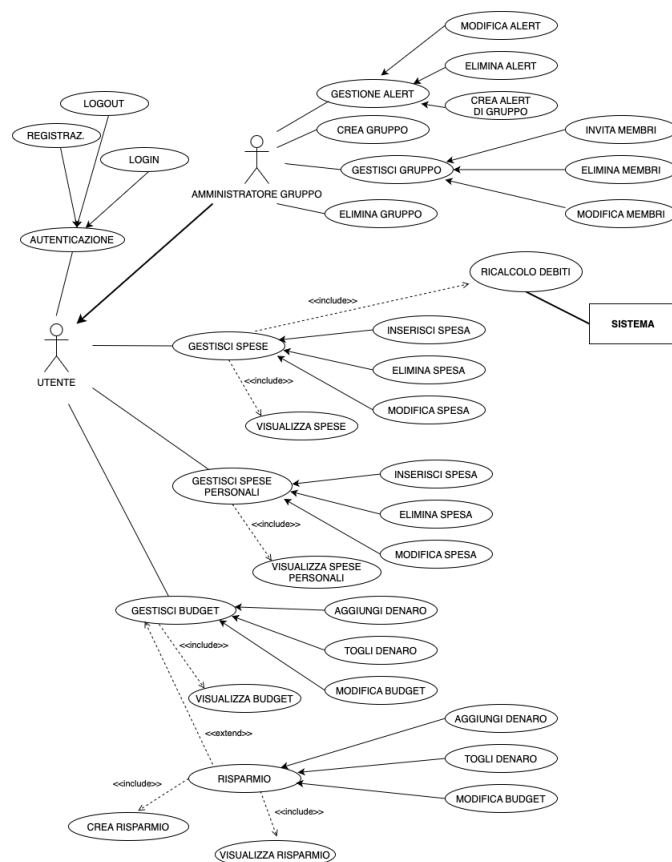


Figura 28: Diagramma dei casi d'uso aggiornato per riflettere l'integrazione relativa alla gestione del risparmio.

4.2.1 Iterazione 3

In questa iterazione sono stati sviluppati i casi d'uso relativi alle spese personali, alla gestione del budget e alla gestione del risparmio. Quest'ultima funzionalità permette all'utente di creare uno o più piani di risparmio con un obiettivo definito, in modo da mettere da parte denaro per esigenze future.

ID	Titolo
UC24	Inserisci spesa personale
UC25	Elimina spesa personale
UC26	Modifica spesa personale
UC27	Visualizza spese personali
UC28	Aggiungi denaro al budget
UC29	Visualizza budget
UC30	Crea risparmio
UC31	Visualizza risparmio
UC32	Modifica risparmio
UC33	Elimina risparmio
UC34	Paga spesa personale

Tabella 6: Iterazione 3 - Casi d'Uso

4.2.2 UC24: Inserimento Spesa Personale

Attori: Utente, Sistema.

Descrizione: Un utente può aggiungere una spesa personale.

Flusso degli eventi:

1. L'utente accede alla sezione delle spese personali.
2. Clicca su "Aggiungi Spesa".
3. Inserisce l'importo, la descrizione e la data.
4. Il sistema registra la spesa e aggiorna il budget.

4.2.3 UC25: Eliminazione Spesa Personale

Attori: Utente, Sistema.

Descrizione: Un utente può eliminare una spesa personale.

Flusso degli eventi:

1. L'utente accede alla lista delle spese personali.
2. Seleziona una spesa e clicca su "Elimina".
3. Il sistema chiede conferma e rimuove la spesa.

4.2.4 UC26: Modifica Spesa Personale

Attori: Utente, Sistema.

Descrizione: Un utente può modificare i dettagli di una spesa personale.

Flusso degli eventi:

1. L'utente accede alla lista delle spese personali.
2. Seleziona una spesa e clicca su "Modifica".
3. Modifica i dati (importo, descrizione, data).
4. Il sistema aggiorna la spesa.

4.2.5 UC27: Visualizzazione Spese Personali

Attori: Utente, Sistema.

Descrizione: Un utente può visualizzare tutte le proprie spese personali.

Flusso degli eventi:

1. L'utente accede alla sezione delle spese personali.
2. Clicca su "Visualizza Spese".
3. Il sistema mostra l'elenco delle spese con dettagli.

4.2.6 UC34: Pagamento Spesa Personale

Attori: Utente, Sistema.

Descrizione: Un utente può saldare una spesa personale direttamente tramite i metodi di pagamento disponibili.

Flusso degli eventi:

1. L'utente accede alla lista delle spese personali.
2. Seleziona una spesa e clicca su "Paga".
3. Il sistema elabora il pagamento e aggiorna il budget scalando l'importo.

4.2.7 UC28: Aggiunta Denaro al Budget

Attori: Utente, Sistema.

Descrizione: Un utente può aggiungere denaro al budget disponibile.

Flusso degli eventi:

1. L'utente accede alla sezione budget.
2. Clicca su "Aggiungi Denaro".
3. Inserisce l'importo.
4. Il sistema aggiorna il budget disponibile.

4.2.8 UC29: Visualizzazione Budget

Attori: Utente, Sistema.

Descrizione: Un utente può visualizzare il budget disponibile e le spese totali.

Flusso degli eventi:

1. L'utente accede alla sezione budget.
2. Il sistema mostra l'importo disponibile e le spese sostenute.

4.2.9 UC30: Creazione Risparmio

Attori: Utente, Sistema.

Descrizione: Un utente può creare un piano di risparmio con un obiettivo definito.

Flusso degli eventi:

1. L'utente accede alla sezione risparmi.
2. Clicca su "Crea Risparmio".
3. Inserisce l'obiettivo e l'importo.
4. Il sistema registra il piano e aggiorna i dati.

4.2.10 UC31: Visualizzazione Risparmio

Attori: Utente, Sistema.

Descrizione: Un utente può visualizzare i piani di risparmio esistenti.

Flusso degli eventi:

1. L'utente accede alla sezione risparmi.
2. Il sistema mostra l'elenco dei risparmi con dettagli.

4.2.11 UC32: Modifica Risparmio

Attori: Utente, Sistema.

Descrizione: Un utente può modificare i dettagli di un piano di risparmio.

Flusso degli eventi:

1. L'utente accede alla lista dei risparmi.
2. Seleziona un piano e clicca su "Modifica".
3. Modifica l'importo o la descrizione.
4. Il sistema aggiorna le informazioni.

4.2.12 UC33: Eliminazione Risparmio

Attori: Utente, Sistema.

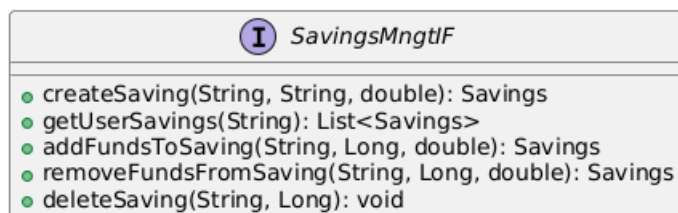
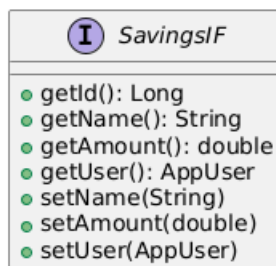
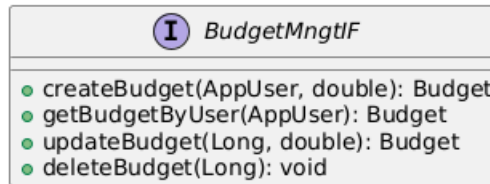
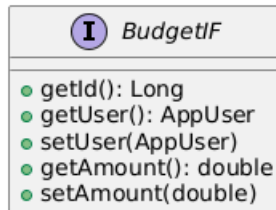
Descrizione: Un utente può eliminare un piano di risparmio.

Flusso degli eventi:

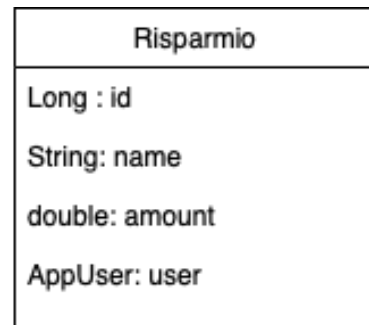
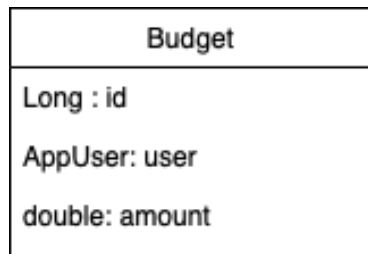
1. L'utente accede alla lista dei risparmi.
2. Seleziona un risparmio e clicca su "Elimina".
3. Il sistema chiede conferma e lo rimuove tornando i soldi allocati al budget.

4.3 Diagramma delle Interfacce

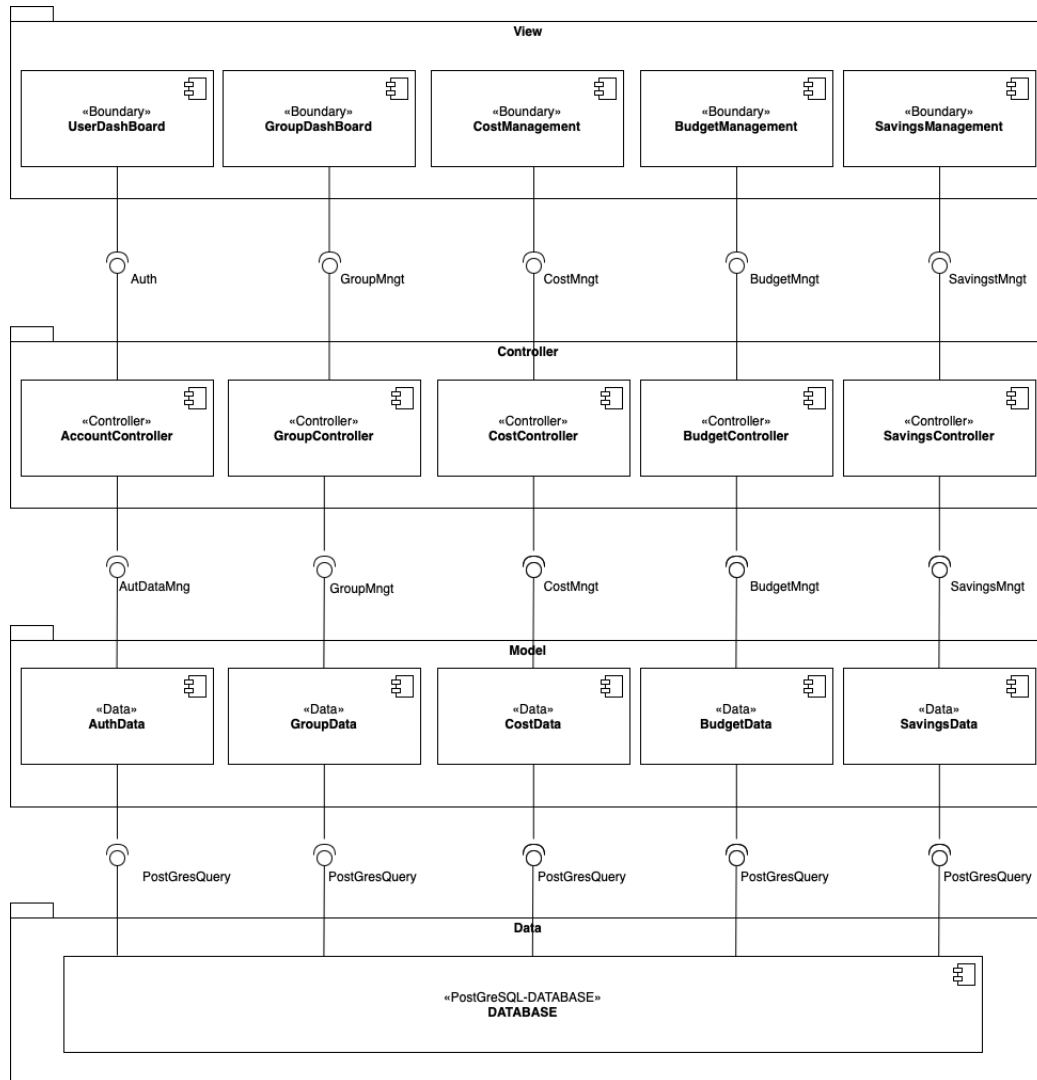
Mostriamo di seguito il diagramma delle interfacce di Budget e Risparmio, implementate in questa iterazione.



4.4 UML Class Diagram per tipi di dato



4.5 UML Component Diagram



Aggiorniamo anche l'UML Component Diagram aggiungendo i componenti relativi alla gestione del Budget e del Risparmio.

4.6 UML Deployment Diagram

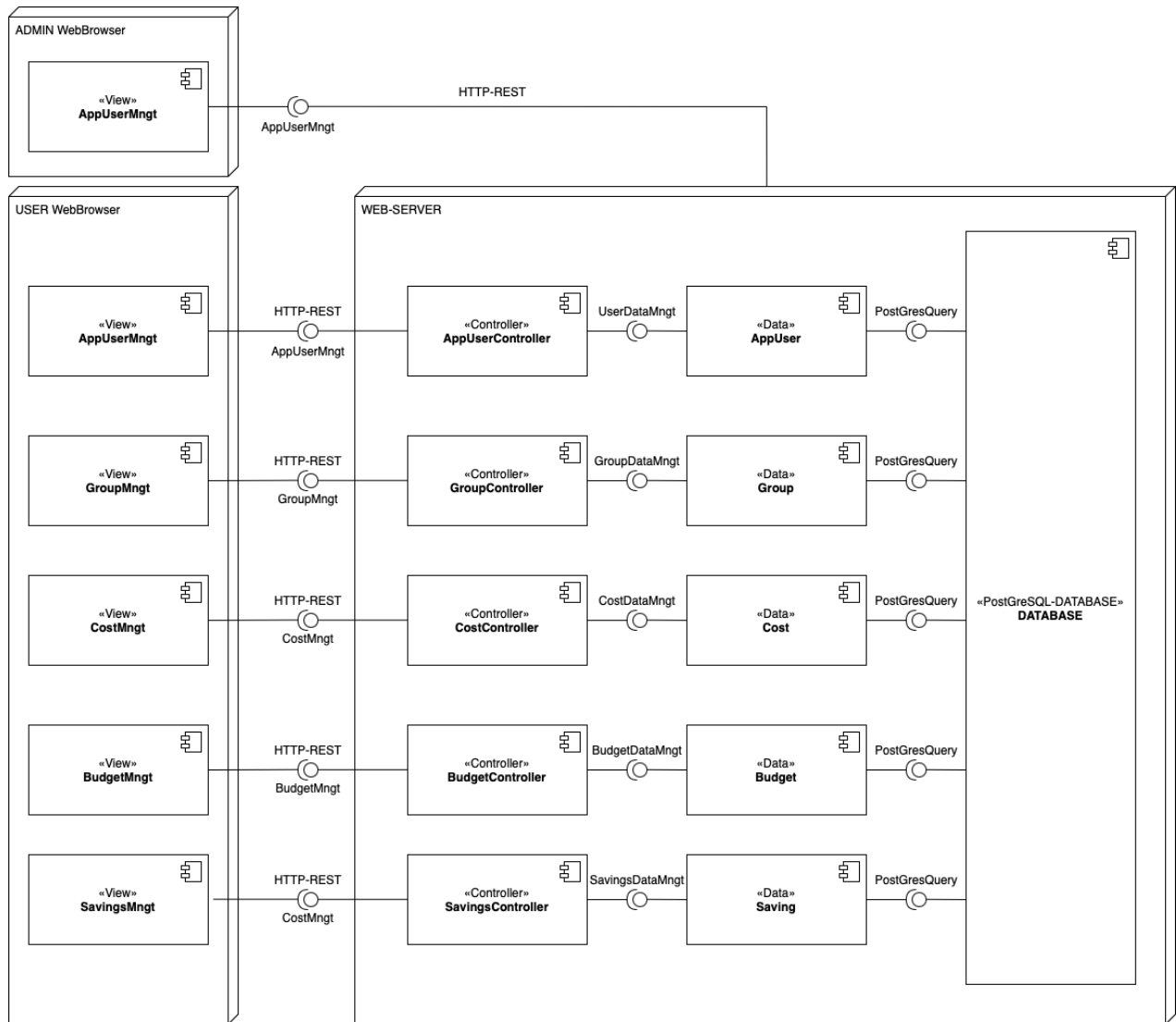


Figura 29: UML Deployment Diagram