Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Laurea triennale in Informatica

Webapp "Biota": piattaforma web per il benessere gastrointestinale

Relatore
Dott. Armir Bujari
Università di Padova

Tutor aziendale
Simone Pozzobon

Laureando
Andrea Trevisin

Alla mia famiglia, che mi ha costantemente supportato in questo lungo percorso, e ai miei amici, con cui ho condiviso studio e vita quotidiana.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli acronimi

API . . . . . Application Programming Interface

AWS . . . . . Amazon Web Services

CoC . . . . . Convention over Configuration

CRUD . . . . . Create, read, Update, Delete

DBMS . . . . . Database Management System

DSL . . . . . Domain Specific Language

DRY . . . . . Don't Repeat Yourself

ERB . . . . . Embedded RuBy

HOTP . . . . . HMAC-based One Time Password

HTTP . . . . . HyperText Transfer Protocol

IDE . . . . . Integrated Development Environment

JSON . . . . . JavaScript Notation Object

MVC . . . . . Model View Controller

OTP . . . . . One Time Password

PDF . . . . . Portable Document Format

REST . . . . . Representational State Transfer

SDK . . . . . Software Development Kit

SNS . . . . . Simple Notification Service

UML . . . . . Unified Modelling Language

URI . . . . . Uniform Resource Identifier

URL . . . . . Uniform Resource Locator

TOTP . . . . . Time-based One Time Password

This is how you do it: you sit down at the keyboard and you put one word after another until its done. It's that easy, and that hard.

Neil Gaiman

1 Introduzione

Il presente documento espone il lavoro svolto dal laureando Andrea Trevisin durante il periodo di stage formativo presso Moku S.r.l. al fine di realizzare il *back end* dell'applicazione web "Biota", una piattaforma per la gestione di esami della flora gastrointestinale. Lo stage, della durata di 300 (trecento) ore, si è svolto tra il 15 maggio e il 17 luglio 2019 presso la sede di Roncade. Il progetto prevedeva il raggiungimento di diversi obiettivi, suddivisi in *milestones*, a partire dalla realizzazione di un *back end* completo delle funzionalità base della piattaforma, per poi implementare i requisiti opzionali e una suite di test completa e infine soddisfare alcuni requisiti desiderabili. Al termine della durata dello stage, la piattaforma è in stato di produzione e tutti gli obiettivi sono stati soddisfatti.

1.1 Descrizione del progetto

Il progetto di stage prevedeva lo sviluppo della parte *back end* dell'applicazione web "Biota". "Biota" è una piattaforma per la gestione di esami della flora gastrointestinale, che permette ai clienti di una farmacia di richiedere un esame mediante la compilazione di un questionario; un campione di materia fecale viene quindi inviato al laboratorio per le analisi, al cui termine viene fornito un referto contenente i risultati delle analisi, la diagnosi del gastroenterologo affiliato e una dieta progettata per migliorare il benessere del paziente. Il *back end* comprende un pannello amministrativo per la gestione dei record del database, permettendone creazione,

distruzione e modifica nonché consentendo operazioni aggiuntive quali il caricamento e lo scaricamento di file per i record designati.

1.2 Azienda

Moku S.r.l. è una startup nata nel 2013, con sede a Roncade (TV), fondata su un progetto software omonimo per una piattaforma web mirata a facilitare il lavoro condiviso su documenti di vario tipo. Si è poi evoluta diventando una società di consulenza IT che realizza prodotti software su commissione. Il team di Moku usa metodologie agili, basate su Scrum, e lavora a stretto contatto con il cliente; questo permette di individuare facilmente e velocemente i requisiti del prodotto, creando prodotti di qualità che rispecchiano le esigenze del committente. I principali ambiti di sviluppo di Moku sono piattaforme web applicazioni Android/IOS.

Figura 1.1: Logo di Moku S.r.l.

1.3 Processi aziendali

1.3.1 Modello di ciclo di vita del software

Il modello di ciclo di vita del software adottato dal team di Moku si basa sulla metodologia Scrum. Scrum è un *framework* agile per la gestione del ciclo di sviluppo del software, iterativo ed incrementale, creato e sviluppato da Ken Schwaber e Jeff Sutherland nel 1995; si propone

di essere leggero, semplice da imparare ma difficile da padroneggiare. Scrum si presta bene alla modalità di lavoro di Moku, in quanto è ideato per team di dimensioni ridotte e permette di gestire progetti complessi offrendo la possibilità di reagire velocemente a cambiamenti. Una delle idee chiave è infatti la "volatilità dei requisiti", ovvero riconoscere che le esigenze dei clienti possano variare in corso d'opera, e che possano sorgere complicazioni non previste - specialmente quando si usano tecnologie all'avanguardia.

L'obiettivo principale è quindi la realizzazione di un prodotto software funzionante e soddisfacente a scapito di aspetti ritenuti non essenziali nell'immediato (e.g. una documentazione completa).

1.3.1.1 Principi della metodologia Scrum

Alla base del *framework* vi è la teoria dei controlli empirici dell'analisi strumentale e funzionale di processo, altrimenti nota come "empirismo", la quale afferma che la conoscenza deriva dall'esperienza, e che le decisioni si devono basare su ciò che si conosce [**?**]. L'implementazione dei controlli empirici di processo si basa su tre concetti:

- **Trasparenza**: gli aspetti significativi del processo devono essere visibili ai responsabili del prodotto; la trasparenza richiede che tali aspetti siano definiti secondo uno standard comune in modo che gli osservatori condividano una comprensione comune di ciò che avviene.
- **Ispezione**: chi usa Scrum deve ispezionare spesso i prodotti della metodologia e il progresso verso l'obiettivo di uno *sprint* per individuare eventuali difformità; tuttavia tale processo non dovrebbe essere frequente al punto da rallentare il lavoro. Le ispezioni danno il massimo beneficio quando eseguite da ispettori qualificati.
- **Adattamento**: se un'ispezione determina che uno o più aspetti sono al di fuori di certi limiti, e che il prodotto risultante è inaccettabile, il processo o il prodotto del processo vanno adattati; l'adattamento deve avvenire il prima possibile per evitare ulteriori difformità.

1.3.1.2 Sprint

Scrum prevede di dividere il progetto in blocchi contigui di lavoro, denominati *sprint*, che prevedono un incremento del prodotto software rilasciabile al termine di ciascuno di essi. Uno *sprint* dura da 1 a 4 settimane, ed è preceduto da una riunione di pianificazione (*sprint planning*) in cui vengono discussi e definiti obiettivi e stimate le tempistiche. Gli obiettivi

stabiliti per uno *sprint* possono variare solo allo *sprint planning* successivo; ogni giorno viene fatta una breve riunione, detta *daily scrum*, in cui viene brevemente discusso il lavoro da svolgere.

Nel corso di uno *sprint*, il team realizza incrementi completi del prodotto: l'insieme di funzionalità inserite in un dato *sprint* sono definite nel *product backlog*, una lista ordinata di requisiti del prodotto che copre l'intero progetto. I requisiti scelti per essere soddisfatti durante un determinato sprint vanno a costituire lo *sprint backlog*.

Gli *sprint* in Moku durano generalmente 10/15 giorni, seguiti da una fase di valutazione del progresso rispetto alle aspettative e di pianificazione dello *sprint* successivo, in cui viene stabilito il nuovo *sprint backlog*. Dato il disaccoppiamento tra *back end* e *front end* nel progetto, i requisiti erano separati in due *backlog differenti*.

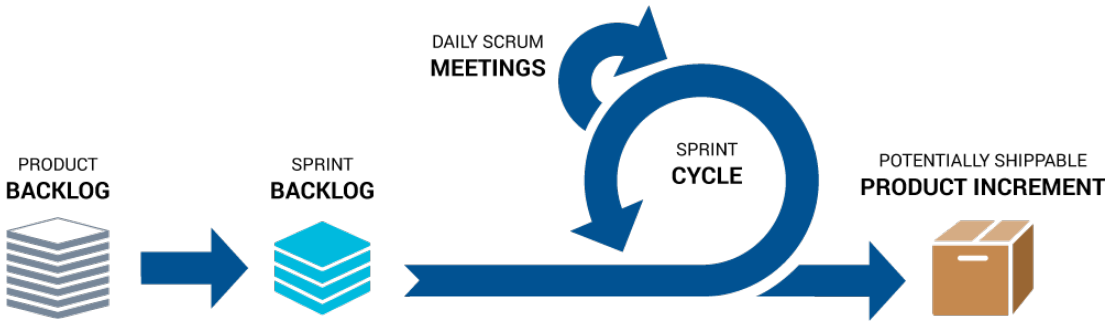

Figura 1.2: Rappresentazione grafica della metodologia Scrum

1.3.1.3 Ruoli di un team Scrum

Il *framework* Scrum prevede che i team siano auto-organizzati, ovvero in grado di scegliere autonomamente il modo migliore per portare a termine il lavoro, e interfunzionali, cioè che non dipendono da altri o da parte del team per realizzare il prodotto. I team Scrum lavorano in maniera iterativa ed incrementale, massimizzando le possibilità di feedback utile.

In un team Scrum sono presenti i seguenti ruoli:

- **Product Owner**: responsabile della massimizzazione del valore del prodotto, rappresenta la "voce" degli stakeholders; definisce gli *items* in base alle *user stories* del cliente, ne assegna la priorità e li aggiunge al *product backlog*.
- **Team di sviluppo**: responsabile della consegna del prodotto, con funzionalità incrementali e rilasciabile al termine di ogni *sprint*; generalmente composto da 3-9 persone con competenze interfunzionali, che realizzano il prodotto effettivo.

- **Scrum Master**: responsabile della rimozione degli ostacoli che intralciano il team di sviluppo nel raggiungimento degli obiettivi dello *sprint*; aiuta il team di sviluppo a comprendere e mettere in pratica la metodologia Scrum, assumendo un ruolo di "*servant/leader*".

Per la durata dello stage, nonostante le linee guida di Scrum lo sconsiglino, i ruoli di *Product Owner* e *Scrum Master* erano ricoperti entrambi dal tutor aziendale per praticità. Il team di sviluppo, composto da due persone, includeva il laureando, il tutor aziendale (che interveniva solo ove fossero necessarie correzioni minori) e uno/due sviluppatori *front end* (a seconda della disponibilità).

1.3.2 Strumenti a supporto dei processi

Per quanto riguarda gli strumenti di supporto ai processi, Moku usa per la maggior parte prodotti appartenenti all'ecosistema Atlassian, ottenendo una maggiore integrazione tra gli strumenti.

1.3.2.1 Project management

Per quanto riguarda la gestione dei progetti, Moku fa affidamento a Jira, un prodotto proprietario di Atlassian. Jira è uno strumento estremamente versatile che offre funzioni di *issue tracking*, *project management* e *bug tracking*. Jira è inoltre fortemente orientato all'utilizzo della metodologia Scrum, permettendo di creare degli *sprint* e di gestire il rispettivo *backlog*.

- **Issue tracking**: in Jira è possibile creare *issues* di tipo *Task* per compiti da completare, *Story* per nuove (*user stories*) e *Feature/Improvement* per funzionalità o miglioramenti a funzionalità esistenti. Ogni *issue* può essere assegnata ad uno o più membri del team, che può aggiornarne il progresso, dividerla in *sub-task* e registrare le ore di lavoro.
- **Enterprise resource planning**: Jira mette a disposizione uno strumento di pianificazione della distribuzione delle risorse temporali ed economiche, sotto il nome di *Tempo*; esso permette di pianificare in anticipo il lavoro da dedicare ad una certa *issue* per ogni membro del team, nonché di tenere traccia dei budget assegnati al progetto.
- **Bug tracking**: in Jira avviene creando issue di tipo *Bug*.

Jira mette a disposizione un'interfaccia drag-and-drop, che ne rende l'uso veloce ed intuitivo.

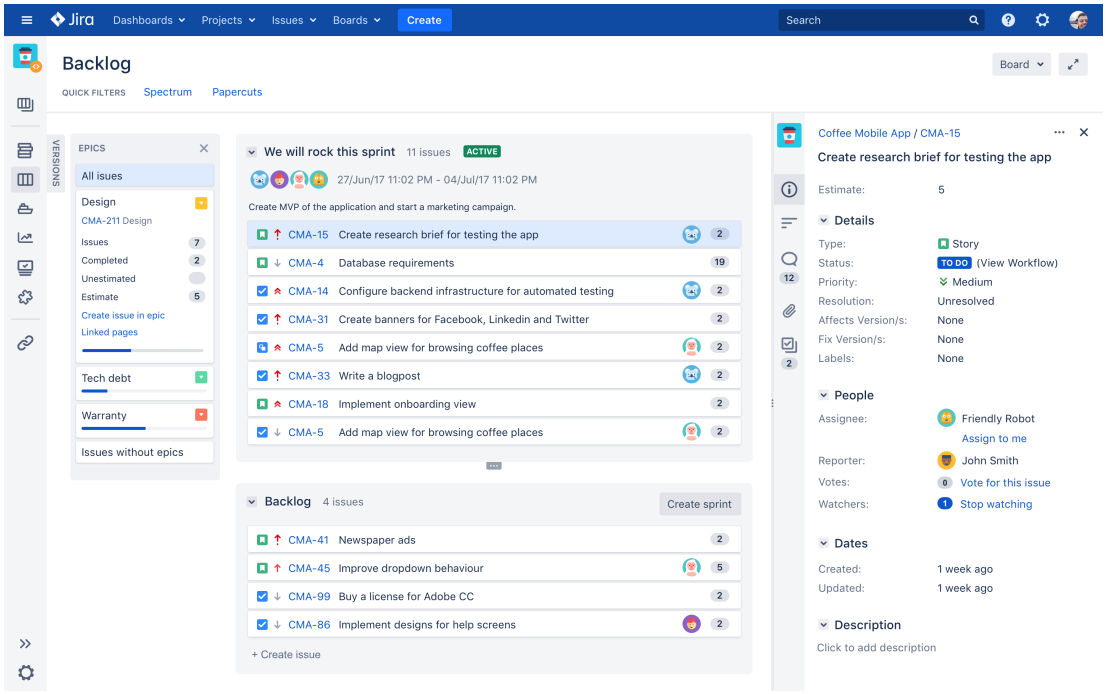

Figura 1.3: Interfaccia della sezione di issue tracking di Jira

1.3.2.2 Gestione del versionamento

Per la gestione del versionamento del prodotto software Moku usa *repository Git* sul servizio di hosting Bitbucket, parte dell'ecosistema Atlassian. Bitbucket offre la possibilità di creare gratuitamente *repository* private, risultando una scelta migliore per l'azienda. Queste vengono gestite localmente tramite il client dedicato Sourcetree, anch'esso proprietario di Atlassian (quindi perfettamente integrato con Bitbucket), il quale semplifica l'uso del paradigma *git-flow*, scelto come standard da Moku.

Git-flow Paradigma di versionamento utilizzato per il progetto, che prevede l'utilizzo sincronizzato di molteplici branch nella repository centrale del prodotto. Il branch **master** contiene l'ultima *release* del prodotto software, ovvero l'ultima versione stabile che include un certo numero di funzionalità testate. I *commit* a questo branch hanno sempre natura incrementale in quanto corrispondono al rilascio di una nuova versione del prodotto. Il branch **develop** invece contiene il prodotto software in lavorazione ed è soggetto a *commit* più frequenti, corrispondenti allo sviluppo di una nuova funzionalità; in ogni caso, in questo branch si trova un prodotto funzionale ma incompleto. Infine è prevista la creazione di vari

branch temporanei, uno per ciascuna *feature* in via di sviluppo. Per aggiungere una *feature* al prodotto, viene effettuato un *fork* del branch **develop** e creato un branch **feature** apposito: al completamento dello sviluppo della funzionalità, viene effettuato il *merge* con il branch di sviluppo. È possibile correggere un errore propagatosi al branch principale con un *hotfix* un *commit* di emergenza. Nel progetto è stato inoltre utilizzato un terzo branch, **staging**, per testare l'integrazione tra *back end* e *front end* prima del rilascio della nuova versione.

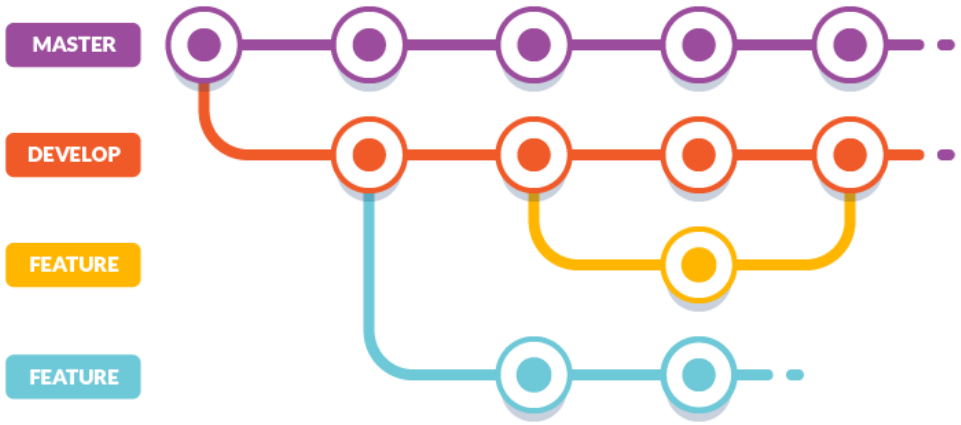

Figura 1.4: Rappresentazione del paradigma *git-flow*

1.3.2.3 Gestione delle comunicazioni

A livello aziendale La corrispondenza riguardante l'avvio del progetto di stage e la collaborazione con il cliente, così come altre comunicazioni a livello aziendale, sono avvenute tramite posta elettronica.

Nel team Per facilitare la comunicazione tra i membri di ogni team di sviluppo, nonché per comunicazioni meno importanti, Moku fa uso di Slack. Slack è un applicativo *cloud-based* di messaggistica e file-sharing, in cui i messaggi vengono inviati in "canali": questo permette di dividere le comunicazioni per progetto, facilitando anche la ricerca di informazioni rilevanti precedentemente inviate.

1.4 Organizzazione del testo

Il primo capitolo presenta una panoramica generale del progetto e dell'ambiente aziendale in cui è stato svolto.
Il secondo capitolo entra in dettaglio nella descrizione del progetto e della sua pianificazione, e introduce le tecnologie con cui è stato realizzato.
Il terzo capitolo espone l'analisi dei requisiti, presentando i casi d'uso individuati e il tracciamento dei requisiti del prodotto.
Il quarto capitolo descrive la progettazione architetturale e di dettaglio dell'applicazione, dando una visione *top-down* del prodotto sviluppato.
Il quinto capitolo presenta i processi atti a garantire la qualità del prodotto.
Il sesto capitolo, infine, raccoglie delle valutazioni oggettive e personali sullo svolgimento e sul risultato dell'esperienza di stage.

1.4.1 Convenzioni tipografiche

Per quanto concerne la stesura di questo documento, sono state adottate le seguenti norme tipografiche:

- Gli acronimi utilizzati sono raccolti in un elenco assieme al loro significato, all'inizio di questo documento;
- I termini in lingua straniera, qualora non facenti parte del comune gergo informatico, sono segnalati in *corsivo*;
- I nomi di file, classi, attributi, metodi e tutto ciò che è classificabile come codice viene riportato con il font `macchina da scrivere`.

If it looks like a duck, and quacks like a duck, we have at least to consider the possibility that we have a small aquatic bird of the family Anatidae on our hands.

Douglas Adams

2 Progetto di stage

2.1 Approfondimento del contesto

2.1.1 Contesto di sviluppo

La piattaforma fa parte di un ecosistema già esistente di servizi per la salute, fruibili presso farmacie affiliate. Esiste, quindi, come parte di un insieme di applicativi paralleli e ne condivide il database e parte della codifica: questo documento si limiterà a descrivere i moduli, le classi e le funzionalità realizzate o adattate per l'implementazione della piattaforma "Biota". L'applicativo si compone di una parte *back end* (in esecuzione su un server dedicato) e una parte *front end* (eseguita all'interno del browser dell'utente) sviluppate separatamente e integrate grazie all'uso di API specifiche; obiettivo dello stage è stata la realizzazione del primo, in modo funzionale all'utilizzo con il secondo. La tecnologia principale, descritta più avanti nel documento, è Ruby on Rails 5 (*framework* per lo sviluppo di applicativi web).

2.1.2 Caratteristiche degli utenti

"Biota" prevede l'utilizzo da parte dii utenti autorizzati con diversi permessi: farmacisti, gastroenterologi e supervisori del laboratorio. Tutte le categorie di utenti autorizzati eseguono il *login* nella piattaforma allo stesso modo, ma ad ognuno viene mostrata un'interfaccia con diverse funzionalità. Il cliente non fa parte degli utenti autorizzati in quanto non può accede-

re direttamente alla piattaforma, e tutte le operazioni che lo coinvolgono sono supervisionate dal farmacista. Per quanto riguarda il pannello amministrativo, ne è previsto l'utilizzo da parte di utenti autorizzati registrati separatamente; tali amministratori fanno parte della società che gestisce la piattaforma. Per interazioni con servizi esterni sono inoltre disponibili delle API REST, utilizzate principalmente per l'integrazione con il gestionale del laboratorio di analisi.

2.1.3 Contesto di utilizzo

Il flusso operativo previsto è il seguente: il cliente richiede al farmacista di effettuare un esame, e inserisce i propri dati registrandosi alla piattaforma; il farmacista quindi ottiene il consenso al trattamento dei dati e fa compilare un breve questionario al cliente, consegnando un kit per il prelievo del campione di materia fecale. Questo viene poi restituito, registrato al cliente ed inviato al laboratorio previa prenotazione della spedizione: grazie alla corrispondenza univoca tra il codice identificativo del campione e la sessione associata all'esame, è garantito l'anonimato del cliente. Il laboratorio, ricevuto il campione, ne effettua il check-in nel proprio gestionale (aggiornandone automaticamente lo stato nella piattaforma) e procede alle analisi necessarie. Viene quindi inviato il referto delle analisi insieme a dei dettagli aggiuntivi, che viene reso disponibile nella piattaforma al gastroenterologo affiliato che aggiunge la sua diagnosi e la dieta consigliata. Dopo un'ulteriore controllo ed eventuali modifiche da parte di un supervisore del laboratorio, il referto completo può essere generato e reso disponibile al cliente *on-demand*, concludendo la procedura.

L'accesso al pannello amministrativo consente di creare, distruggere e modificare alcune categorie di record del database, come ad esempio le varie tipologie di utenti o le farmacie affiliate. Esso viene utilizzato principalmente per l'inserimento di nuovi utenti autorizzati, il recupero di informazioni rilevanti quali il consenso al trattamento dei dati, e per la risoluzione di problematiche grazie all'accesso privilegiato al database.

2.2 Definizione degli obiettivi

Nella seguente sezione verranno esposti in via generale gli obiettivi e requisiti che l'applicativo realizzato doveva soddisfare. È importante notare che, data la natura agile del metodo di lavoro impiegato dall'azienda, gli obiettivi presentati nel piano di lavoro sono stati soggetti a modifiche e aggiunte di lieve e media entità; per completezza e possibilità di confronto, di

seguito verranno esposti sia i requisiti iniziali che quelli elaborati a fronte dell'interazione con il cliente.

2.2.1 Notazione

Si farà riferimento agli obiettivi definiti secondo la seguente notazione:

- **O** indica un obiettivo obbligatorio, vincolante in quanto requisito primario stabilito dal committente.
- **D** indica un obiettivo desiderabile, non vincolante o strettamente necessario, ma dal riconoscibile valore aggiunto.
- **F** indica un obiettivo facoltativo, rappresentante un valore aggiunto non necessariamente competitivo.

2.2.2 Piano di lavoro

Il piano di lavoro, stilato in collaborazione con il relatore ed il tutor aziendale nella settimana precedente all'inizio dello stage, espone una serie di requisiti emersi dalle esigenze del cliente emerse in fase di avvio del progetto.

2.2.2.1 Obiettivi

Obbligatori

- O01: Analisi ed implementazione API REST laboratorio
- O02: Analisi ed implementazione API GraphQL per il flusso operativo base

Desiderabili

- D01: Analisi ed implementazione di test dei sistemi di pagamento e fatturazione
- D02: Suite di testing del software prodotto
- D03: Documentazione completa

Facoltativi

- F01: Ulteriori modifiche all'applicazione che esulano da quando riportato nel piano di lavoro

2.2.3 Modifiche al piano

Durante il corso dello stage, nello svolgimento del ruolo di Product Owner da parte del tutor aziendale, ci sono state varie interazioni con il cliente che hanno portato a ridefinire alcuni requisiti desiderabili e a specificare in modo più preciso i requisiti facoltativi. Tali modifiche sono sempre state fatte in accordo alle possibilità di soddisfazione entro i tempi previsti, e tenendo in mente che la piattaforma sarebbe entrata in produzione al completamento delle funzionalità per l'esecuzione flusso di attività base.

2.2.3.1 Obiettivi rivisti

Obbligatori

- O01: Analisi ed implementazione API REST laboratorio
- O02: Analisi ed implementazione API GraphQL per il flusso operativo base
- O03: Adattamento e rafforzamento del codice esistente

Desiderabili

- D01: Generazione PDF con informazioni specifiche
- D02: Suite di testing del software prodotto
- D03: Documentazione completa
- D04: Registrazione del consenso al trattamento dei dati

Facoltativi

- F01: Autenticazione mediante HMAC
- F02: Ulteriori modifiche all'applicazione

2.3 Pianificazione delle attività

Il piano di lavoro prevedeva anche una suddivisione settimanale delle attività dedicate alla realizzazione del progetto, distribuendo le 300 ore previste su un periodo di 8 settimane, più una settimana di *slack* per accomodare eventuali impegni accademici.

La suddivisione settimanale prevista era la seguente:

- **Prima settimana (20 ore)**: Comprensione del sistema e degli obiettivi;
- **Seconda settimana (40 ore)**: Analisi dei requisiti;
- **Terza settimana (40 ore)**: Studio e setup ambiente di sviluppo, progettazione;
- **Quarta settimana (40 ore)**: Implementazione;
- **Quinta settimana (40 ore)**: Implementazione;
- **Sesta settimana (40 ore)**: Implementazione;
- **Settima settimana (40 ore)**: Implementazione, test e validazione;
- **Ottava settimana (40 ore)**: Test e validazione, documentazione.

Attività	Monte ore
Comprensione sistema e obiettivi	20
Analisi dei requisiti	40
Progettazione	20
Studio e setup ambiente di sviluppo	20
Implementazione	150
Test e validazione	30
Documentazione	20
Totale	300

Tabella 2.1: Totale di ore dedicato a ciascuna attività.

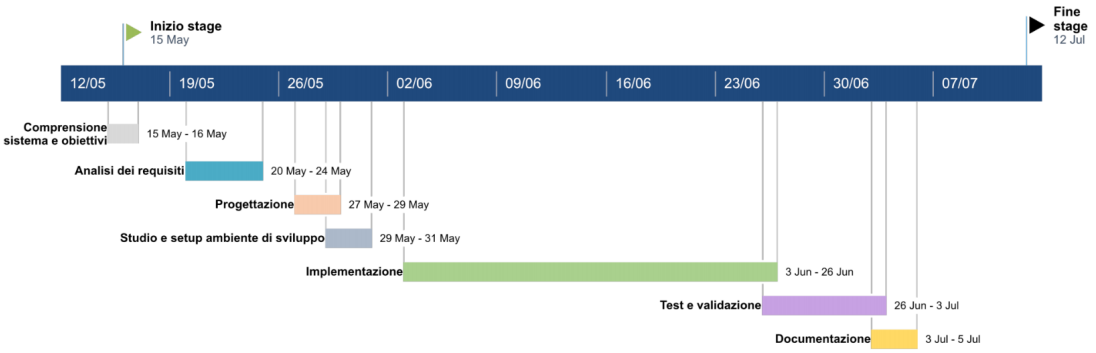

Figura 2.1: Diagramma di Gantt di pianificazione delle attività.

2.4 Tecnologie utilizzate

Di seguito vengono discusse le tecnologie individuate per la realizzazione del progetto. Alcune di esse, quali Ruby on Rails e RubyMine, sono state richieste dell'azienda; altre (principalmente gemme per Ruby) sono state ricercate e proposte dal laureando per soddisfare alcuni dei requisiti individuati.

2.4.1 Ruby

Figura 2.2: Logo di Ruby.

Ruby è un linguaggio di programmazione ad alto livello, interpretato, e orientato agli oggetti con paradigma puro (ogni componente del linguaggio è trattato come un oggetto)[**?**].

Alcune tra le caratteristiche più rilevanti di Ruby sono la presenza di tipizzazione dinamica, *garbage collector* e *duck typing* ("Se sembra un'anatra, nuota come un'anatra e starnazza come un'anatra, allora probabilmente è un'anatra."), ovvero il poter considerare l'insieme dei metodi di un oggetto anziché il suo tipo per decidere se è valido a *run-time*.

Si tratta un linguaggio molto flessibile che offre grande libertà allo sviluppatore e supporta pratiche come il *monkey patching* (ridefinire una classe in un punto diverso dalla definizione

originale) e la ridefinizione di metodi a *run-time*. Esiste una grande varietà di programmi e librerie Ruby, noti come "gemme", il cui utilizzo verrà discusso in seguito.

2.4.2 Gemme

Le gemme sono librerie e programmi Ruby distribuiti sotto forma di pacchetti in modo non dissimile dai moduli in Node.js: l'installazione è gestita dal *package manager* RubyGem, individualmente tramite terminale (con il comando `gem install nome_gemma`) oppure di gruppo specificando le gemme desiderate nel file `Gemfile`, che viene automaticamente letto da RubyGem con l'esecuzione del comando `bundle install`.

2.4.2.1 AWS SDK for Ruby

Insieme di gemme che facilitano l'interazione con i servizi web di Amazon, fornendo classi e metodi ad-hoc; l'SDK è suddiviso in moduli specifici per ogni servizio, permettendo di scegliere quali gemme usare a seconda delle necessità, e di aggiornare le gemme utilizzate in modo indipendente. In particolare è stata utilizzata la gemma `aws-sdk-sns` per l'invio di SMS tramite Amazon Simple Notification Service.

2.4.2.2 Database Cleaner

Gemma di supporto all'esecuzione dei test, offre varie strategie di pulizia di un database per il testing di una applicazione in Ruby. Il caso d'uso principale di questa gemma è l'assicurare di avere un database pulito prima di eseguire i test e di non avere record residui dopo ciascun test, in modo da non avere interferenze nei risultati. Supporta il DBMS PostgreSQL.

2.4.2.3 Factory Bot

Gemma di supporto alla realizzazione di test, permette di definire la procedura di creazione di oggetti complessi attraverso metodi a cui passare pochi parametri (detti *factory*). Definire una *factory* per una classe consente di richiamarla in seguito per generare nuove istanze di quella classe, con attributi generati casualmente o specificati manualmente. Factory Bot offre molte altre funzionalità, quali la possibilità di definire vari *trait* (collezioni di attributi assegnati in un modo specifico) per la creazione di un oggetto e di introdurre ereditarietà tra *factories*. È stato scelto di usare questa gemma in quanto permette di realizzare facilmente istanze di *models* di Rails replicandone anche le relazioni con altri *models*.

2.4.2.4 HexaPDF

Libreria che permette interazioni ad alto e basso livello con file PDF: è possibile leggere e modificare il codice sorgente di un documento, oppure sfruttare i *wrappers* presenti per effettuare operazioni ad alto livello come l'inserimento di immagini o figure. Utilizzata per la generazione dinamica di referti.

2.4.2.5 Jbuilder

Jbuilder è una gemma molto utile che fornisce un DSL per dichiarare strutture JSON anche complesse, utilizzando strutture *if/else* e cicli. Un file `.jbuilder` contiene la dichiarazione di una struttura dati JSON che, al momento del *rendering* da parte di un *controller* di Ruby on Rails, viene elaborata e generata secondo i dati disponibili. Questa gemma è stata utilizzata per generare i body delle risposte HTTP date dai *controller*, data la necessità di utilizzare strutture ricorsive.

2.4.2.6 ROTP

Acronimo di "Ruby One Time Password", questa gemma permette di generare e verificare codici HOTP e TOTP in accordo agli standard RFC 4426 [**?**] e RFC 6238 [**?**]. ROTP è inoltre compatibile con Google Authenticator su dispositivi Android e iOS. La gemma è stata sfruttata per verificare il consenso al trattamento dei dati mediante codici TOTP inviati per SMS.

2.4.2.7 RSpec

Framework di *behaviour-driven development* per Ruby, RSpec è un *tool* per la creazione di test per codice Ruby. RSpec fornisce un DSL molto intuitivo e descrittivo per la realizzazione dei test, permettendo di definire precisamente i vari scenari in cui si desidera verificare il comportamento del codice. Il funzionamento di base di RSpec si basa su "esempi" e "aspettative": un esempio è un particolare scenario che si desidera testare, e le aspettative descrivono lo stato che il codice dovrebbe raggiungere ad un certo punto della sua esecuzione. RSpec, in congiunzione con altre librerie di supporto, è stato utilizzato per realizzare la suite di test per l'applicativo.

2.4.2.8 RuboCop

Gemma per l'analisi statica e la formattazione del codice, basata sulla *style guide* della community di Ruby. RuboCop permette di scansionare il codice rilevando errori e segnalandoli allo sviluppatore; per errori comuni è anche disponibile una funzione di correzione automatica.

2.4.2.9 Shoulda Matchers

Gemma di supporto alla creazione di test, semplifica di molto la creazione di nuovi esempi con RSpec fornendo una sintassi più sintetica per le aspettative. È stata utilizzata nel progetto per rendere i test più leggeri, leggibili e veloci da realizzare.

2.4.3 Ruby on Rails

Figura 2.3: Logo di Rails.

Per la realizzazione del progetto Moku ha scelto di usare Ruby on Rails 5 (noto anche come Rails), un *framework open source* per applicativi web realizzato in Ruby e utilizzato da celebri siti come GitHub (servizio di hosting per *repository* Git), Twitch (servizio di *live streaming*) e SoundCloud (piattaforma di distribuzione musicale)[**?**]. Rails è stato scelto per la caratteristica di velocizzare notevolmente lo sviluppo di nuovi applicativi, rimuovendo le parti "ripetitive": ad esempio offrendo alias concisi per operazioni di base (e.g. iterazioni su collezioni di oggetti, strutture if/else) che riducono la verbosità del codice.

I principi cardine di Rails sono due:

- **Do not Repeat Yourself**: il principio DRY sostiene che vadano evitate tutte le forme di ripetizione e ridondanza logica nell'implementazione del software [**?**]; ad esempio, non è necessario specificare le colonne della tabella del database nella definizione di una classe, in quanto Rails recupera automaticamente tale informazione.
- **Convention over Configuration**: il principio CoC sostiene che il programmatore dovrebbe esplicitare solo le parti "non convenzionali" del codice; ad esempio in Rails esiste per convenzione una corrispondenza tra il nome di una classe e il nome di una tabella del database, quindi essa non va specificata.

2.4.3.1 Integrazione tra i componenti

Rails è un *framework full-stack*, ovvero offre tutti i componenti richiesti per lo sviluppo di un applicativo web, nativamente integrati tra di loro: tramite l'uso di script per la creazione di file, detti *generators*, è possibile creare contemporaneamente sia una tabella del database che la classe corrispondente; essi sono automaticamente associati tramite convenzioni di nomenclatura.

Il database, indipendentemente dall'implementazione, può essere modificato tramite *migrations*, istruzioni per la modifica dello schema da parte di Rails. Le *migrations* sono scritte in un DSL in Ruby e versionate automaticamente, permettendo di effettuare un *rollback* ad una versione precedente dello schema.

2.4.3.2 Pattern architetturale

Gli applicativi realizzati in Rails seguono necessariamente un pattern *MVC*.

Model Un *model* è una classe associata ad una tabella del database secondo uno standard convenzionale: una tabella corrisponde una classe, le colonne sono convertite in attributi e le righe rappresentate come istanze della classe. I *models* possono venire generati automaticamente dalla definizione della tabella. Oltre ai normali vincoli di database, in Rails è possibile definire ulteriori controlli sui valori in database direttamente nel *model*. La filosofia di Rails prevede che essi contengano la quasi interezza della *business logic*.

Controller I *controllers* sono componenti che rispondo alle richieste del server web, determinando quale *view* caricare; essi possono anche interrogare i *models* per mostrare informazioni aggiuntive, e rendere disponibili "azioni" per fare richieste al server. I controller sono resi disponibili mediante il file di *routing* `routes.rb`, in cui vengono associati a delle specifiche richieste; Rails incoraggia gli sviluppatori all'uso di *RESTful routes*, che includono azioni come "index", "new", "show", "edit".

View Di default le *views* sono file con estensione `.erb` contenenti codice HTML misto a codice Ruby, che vengono elaborati a *run-time* permettendo una visualizzazione dinamica delle informazioni. In alternativa, per esempio nell'implementazione di una RESTful API, una *view*può essere un file JSON contenente il *body* di una risposta.

2.4.3.3 Devise

Framework di autenticazione utenti per Rails. Si tratta di una gemma altamente modulare e flessibile, composta 10 sotto-moduli, ognuno dei quali implementa una data funzionalità, permettendo di "comporre" a proprio piacimento un sistema di autenticazione con le caratteristiche desiderate (e.g. recupero password, tracciamento di orari e indirizzi IP per ogni accesso e validazione tramite e-mail). Devise è stato utilizzato per l'autenticazione sulla piattaforma da parte degli utenti registrati.

2.4.3.4 ActiveAdmin

ActiveAdmin è un *framework* per la generazione di interfacce amministrative per applicativi web realizzati con Rails. ActiveAdmin astrae pattern ricorrenti per automatizzare la generazione di elementi comuni dell'interfaccia, ad esempio operazioni quali la creazione, visualizzazione o modifica di oggetti appartenenti a uno o più *models*. ActiveAdmin si integra con la configurazione presente di Devise per gestire l'autenticazione. In ActiveAdmin viene fornita un'interfaccia di default pienamente configurabile: è possibile aggiungere un pannello relativo ad una tabella particolare del database, applicarvi filtri, e decidere quali campi rendere disponibili alle operazioni di visualizzazione e modifica.

2.4.4 RubyMine

Figura 2.4: Logo di RubyMine.

Ambiente di sviluppo integrato multipiattaforma, progettato specificamente per Ruby on Rails e realizzato da JetBrains. RubyMine mette a disposizione vari strumenti per facilitare lo sviluppo, tra cui completamento automatico del codice, linting avanzato con calcolo della complessità ciclomatica, e creazione guidata di *migrations* e *generators*. Altre caratteristiche che hanno portato a sceglierlo come IDE di riferimento sono la possibilità di testare ed eseguire l'applicativo in locale, il controllo di versione integrato (comodo per cambiare il *branch* corrente senza dover utilizzare Sourcetree) e lo strumento di *refactoring* che rispetta

le convenzioni di Rails (e.g. modificando il nome di un *model*, i rispettivi *controller* e *view* vengono aggiornati.)

2.4.5 GraphQL

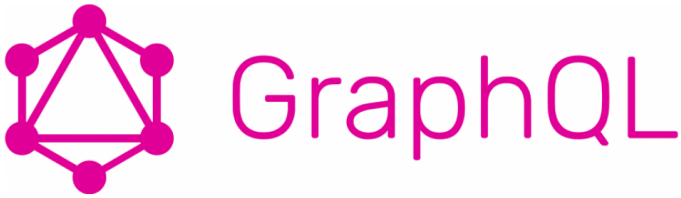

Figura 2.5: Logo di GraphQL.

Linguaggio di query *open source*, sviluppato da Facebook nel 2012 e disponibile al pubblico dal 2015. La sintassi di GraphQL è molto simile a quella del formato JSON, pensata per una lettura più facile da parte di operatori umani. La particolarità di GraphQL è che include anche il *runtime system* e il sistema dei tipi, quindi non dipende dalla specifica implementazione del database. I tipi sono definiti dallo sviluppatore e vengono utilizzati da GraphQL per validare le richieste e respingere query errate.

GraphQL offre due tipi di operazioni: query, semplici interrogazioni al database, e *mutations*, ovvero operazioni di modifica del database o interazioni particolari (autenticazione, esecuzione di un comando).

Il *runtime system* di GraphQL, eseguito sul server, è responsabile della validazione delle richieste e della serializzazione delle risposte in formato JSON. Sono disponibili librerie per creare API in vari linguaggi di programmazione, tra cui Ruby.

Nel progetto GraphQL è stato usato per implementare l'API di comunicazione con il *front end* della piattaforma, mettendo a disposizione varie query rilevanti nonché *mutations* per autenticazione, interazione con il database e richieste di elaborazione al *back end*.

2.4.5.1 Altair

Ambiente di sviluppo integrato *open source* e multipiattaforma per GraphQL. Altair fornisce una semplice interfaccia per testare API GraphQL, mettendo a disposizione funzioni utili quali formattazione automatica,generazione automatica della documentazione del sistema dei tipi e aggiunta di *headers* alla richiesta. Altair è stato utilizzato per testare le API GraphQL prima di renderle disponibili al *front end*.

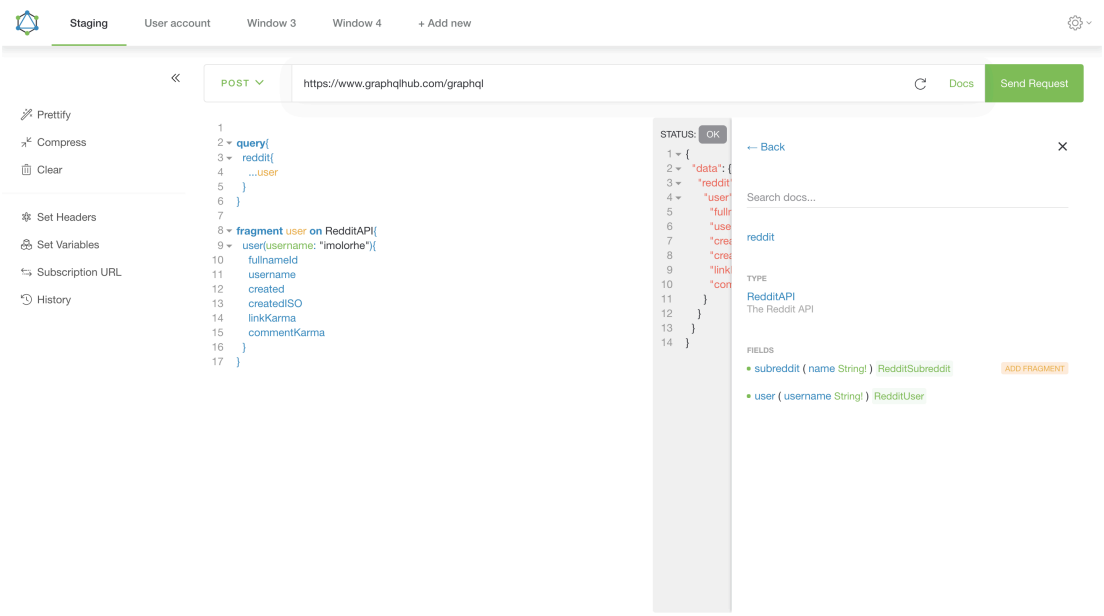

Figura 2.6: Interfaccia di Altair.

2.4.6 HTTP

Noto protocollo a livello applicativo per la trasmissione di informazioni in una rete. Il protocollo HTTP prevede un'architettura di tipo *client/server*, in cui il *client* esegue una richiesta e il *server* la elabora, fornendo una risposta adeguata.

Una richiesta HTTP si compone di quattro parti:

- una "riga di richiesta" che specifica il tipo di operazione e l'URI dell'oggetto della richiesta;
- una "sezione *header*" in cui vengono fornite informazioni aggiuntive (e.g. credenziali di autenticazione);
- una riga vuota che contiene i due caratteri *carriage return* e *line feed*;
- un *body*, ovvero il corpo del messaggio.

Il messaggio di risposta segue una struttura simile, con l'eccezione della riga di richiesta che viene sostituita da una "riga di stato",contenente informazioni sul risultato della richiesta sotto forma di codici standard.

Il protocollo HTTP è largamente utilizzato per l'implementazione di RESTful API; per quanto riguarda il progetto, è stato utilizzato per realizzare l'API che si sarebbe interfacciata

con il software gestionale del laboratorio di analisi, esponendo determinati *endpoint* per le richieste.

2.4.6.1 Insomnia

Figura 2.7: Logo di Insomnia.

Client *open source* e multipiattaforma per API GraphQL e REST, consente di gestire facilmente lo sviluppo e il testing di richieste HTTP e GraphQL; offre funzionalità di formattazione automatica, variabili d'ambiente e supporto a vari tipi di autenticazione. Insomnia è stato utilizzato per testare gli endpoint delle API per il laboratorio di analisi, nonché per sviluppare un client per le API del corriere SDA.

2.4.7 PostgreSQL

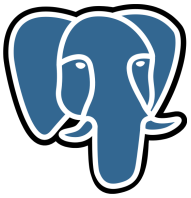

Figura 2.8: Logo di PostgreSQL.

DBMS gratuito ed *open source* per la gestione di database relazionali, è considerato uno dei migliori DBMS gratuiti per la realizzazione di applicazioni web. Oltre alle *feature* di base che condivide con altre soluzioni simili, come l'utilizzo del linguaggio SQL per eseguire query sui dati, PostgreSQL offre delle caratteristiche che risultano molto utili: principalmente offre un sistema di tipi in cui è possibile definire tipi più complessi a partire da quelli di *default* del linguaggio SQL; permette inoltre l'ereditarietà dei tipi, facilitando un approccio *object-oriented* alla progettazione del database.

PostgreSQL è stato scelto per questo progetto non tanto per le sue caratteristiche peculiari quanto per la sua robustezza e flessibilità; la gestione del database, una volta configurato per funzionare con Rails, è comunque delegata a quest'ultimo.

You insist that there is something a machine cannot do. If you tell me precisely what it is a machine cannot do, then I can always make a machine which will do just that.

John von Neumann

3 Analisi dei requisiti

3.1 Casi d'uso

Di seguito verranno descritti i casi d'uso dell'applicazione, per quanto riguarda le funzionalità da me progettate; non verranno quindi tenute in considerazione eventuali funzionalità pre-esistenti che fossero parte dell'ecosistema a cui appartiene la piattaforma.

I casi d'uso verranno inoltre considerati per quanto concerne il *back end* dell'applicazione, ovvero sulla base delle funzionalità realizzate per essere utilizzate dal *front end*: eventuali aggiunte, quali la visualizzazione di messaggi di errore dettagliati o reindirizzamenti a pagine differenti della piattaforma web, verranno ignorate.

I casi d'uso verranno descritti mediante dei diagrammi UML che rappresentano le possibili interazioni tra gli **attori** ed il **sistema** dal punto di vista dei primi. È importante notare che il sistema, ovvero il *back end* sviluppato, non viene utilizzato direttamente dagli utenti finali ma risponde a richieste di servizi esterni come il *front end* (con l'eccezione degli utenti amministrativi, a cui viene fornita un'interfaccia di interazione ad-hoc): i casi d'uso verranno dunque trattati di conseguenza.

Di seguito verranno elencati solo i casi d'uso principali; per quelli secondari consultare l'appendice A.

3.1.1 Organizzazione dei casi d'uso

3.1.1.1 Attori

- **Utente non autenticato**: un utente che deve effettuare l'autenticazione per accedere alle funzionalità disponibili al suo ruolo;
- **Amministratore**: un utente autenticato con privilegi di accesso al pannello amministrativo.
- **Front end**: una istanza del *front end* della piattaforma;
- **Gestionale laboratorio**: il software gestionale del laboratorio;

3.1.1.2 Struttura

Per ogni caso d'uso, identificato da un codice univoco, vengono specificati gli attori coinvolti, lo scopo da raggiungere, le azioni previste e le condizioni del sistema prima e dopo tali azioni. I casi d'uso verranno descritti dalla seguente struttura:

- Codice identificativo: **UC {codice_padre}.{codice_figlio}**
  - **UC** indica che si tratta di un caso d'uso;
  - **codice_padre** identifica il caso d'uso principale a cui si fa riferimento;
  - **codice_figlio** identifica il sottocaso specifico;
- Titolo
- Diagramma UML
- Attori
- Attori secondari, se presenti
- Scopo e descrizione
- Precondizioni
- Scenario principale
- Postcondizioni
- Inclusioni (se presenti)
- Estensioni (se presenti)

3.1.1.3 Casi d'uso generali

Per una miglior suddivisione, le operazioni che avvengono sotto condizioni molto simili (principalmente quelle che possono essere richieste da un utente autenticato con un determinato ruolo) saranno raggruppate in dei casi d'uso di alto livello, d'ora in poi "casi d'uso generali", identificati dalla lettera **G** prefissa al loro codice numerico. Tali casi d'uso servono solo per una miglior comprensione, quindi il loro codice non verrà considerato come **codice_padre** nell'identificazione dei casi d'uso raggruppati.

3.1.2 UC1: Autenticazione amministratore

Figura 3.1: Diagramma UML per UC1 con estensioni

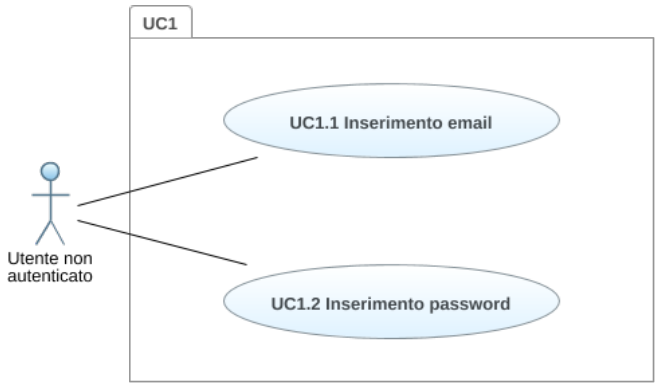

Figura 3.2: Diagramma UML per UC1 con sottocasi

Attori: Utente non autenticato.
Scopo e descrizione: L'attore vuole effettuare la procedura di autenticazione al pannello di amministrazione.
Precondizioni: L'applicativo è stato avviato con successo e l'attore ha accesso alla pagina

di autenticazione.
Scenario principale:

- L'attore inserisce un indirizzo email valido (UC 1.1);
- L'attore inserisce la password associata a tale indirizzo email (UC 1.2);
- L'attore clicca sul pulsante di *login*.

Postcondizioni: L'attore viene autenticato dal sistema e ha accesso al pannello di amministrazione.
Estensioni: L'attore non viene autenticato e visualizza un messaggio di errore (UC2).

3.1.3 UC2: Errore autenticazione amministratore

Attori: Utente non autenticato.
Scopo e descrizione: L'attore vuole effettuare la procedura di autenticazione al pannello di amministrazione.
Precondizioni: L'attore ha inserito le proprie credenziali negli appositi campi.
Scenario principale:

- L'attore clicca sul pulsante di *login*;
- L'attore visualizza un messaggio di errore che lo informa che le credenziali non sono corrette.

Postcondizioni: L'attore non viene autenticato dal sistema e rimane nella pagina di autenticazione.

3.1.4 UC3: Autenticazione front end

Attori: Front end.
Scopo e descrizione: L'attore vuole effettuare la procedura di autenticazione per un utente della piattaforma.
Precondizioni: L'applicativo è stato avviato con successo e l'attore è collegato correttamente al sistema.
Scenario principale: L'attore invia una richiesta di autenticazione contenente le credenziali di accesso di un utente autorizzato.

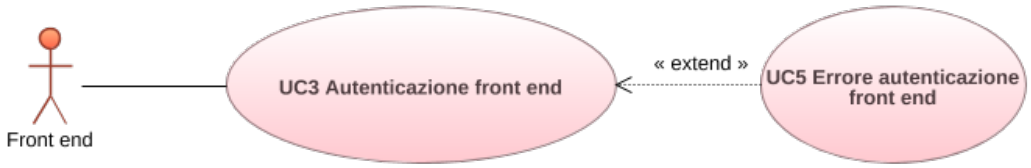

Figura 3.3: Diagramma UML per UC3

Postcondizioni: L'utente specificato dall'attore viene autenticato, e l'attore ha ricevuto in risposta i parametri della relativa sessione attiva.
Estensioni: L'utente specificato dall'attore non viene riconosciuto, e l'attore ha ricevuto in risposta un messaggio di errore che lo informa che le credenziali non sono corrette (UC5).

3.1.5 UC4: Reset password per accesso alla piattaforma

Attori: Front end.
Scopo e descrizione: L'attore vuole effettuare la procedura di reset della password per un utente della piattaforma.
Precondizioni: L'applicativo è stato avviato con successo e l'attore è collegato correttamente al sistema.
Scenario principale: L'attore invia una richiesta di reset della password di accesso di un utente autorizzato.
Postcondizioni: È stata generata ed inviata all'utente una nuova password,e l'attore ha ricevuto una risposta di conferma del successo dell'operazione.

3.1.6 UC5: Errore autenticazione front end

Attori: Front end.
Scopo e descrizione: L'attore vuole effettuare la procedura di autenticazione per un utente della piattaforma.
Precondizioni: L'attore ha inviato una richiesta di autenticazione al sistema.
Scenario principale:

- L'attore è in attesa di una risposta dal sistema;
- L'attore riceve in risposta un messaggio di errore che lo informa che le credenziali non sono corrette.

Postcondizioni: L'utente specificato dall'attore non viene riconosciuto e non ha ricevuto i dati di una sessione valida.

3.1.7 UCG1: Operazioni amministrative

Questo caso d'uso generale riassume le operazioni disponibili ad un utente autorizzato che abbia eseguito l'accesso al pannello di amministrazione. La maggior parte delle operazioni sui record sono identiche tra loro, e verranno quindi riportate in modo generalizzato; eventuali operazioni specifiche verranno contestualizzate nel proprio caso d'uso.

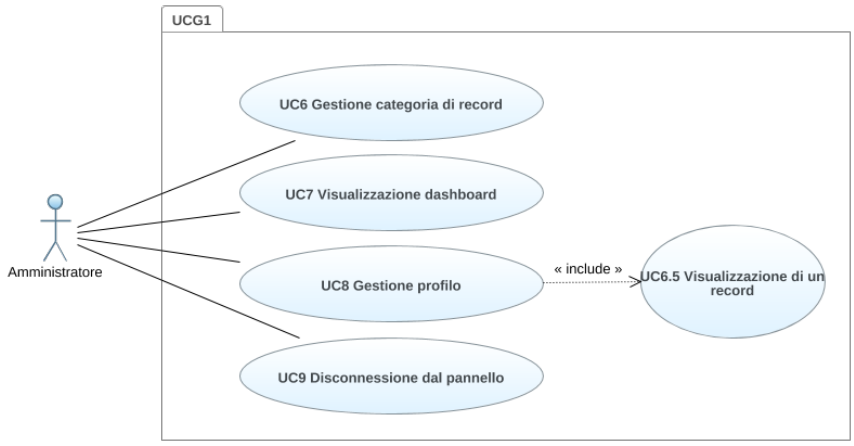

Figura 3.4: Diagramma UML per UCG1

Attori: Amministratore.
Scopo e descrizione: L'attore vuole eseguire operazioni di amministrazione della piattaforma.
Precondizioni: L'attore ha ottenuto l'accesso al pannello di amministrazione.
Scenario principale:

- L'attore visualizza e gestisce una categoria di record (UC6);
- L'attore visualizza la *dashboard* (UC7);
- L'attore gestisce il proprio profilo (UC8);
- L'attore effettua la disconnessione dal pannello di amministrazione (UC9).

Postcondizioni: L'attore ha correttamente portato a termine le operazioni desiderate.

3.1.7.1 UC6: Gestione categoria di record

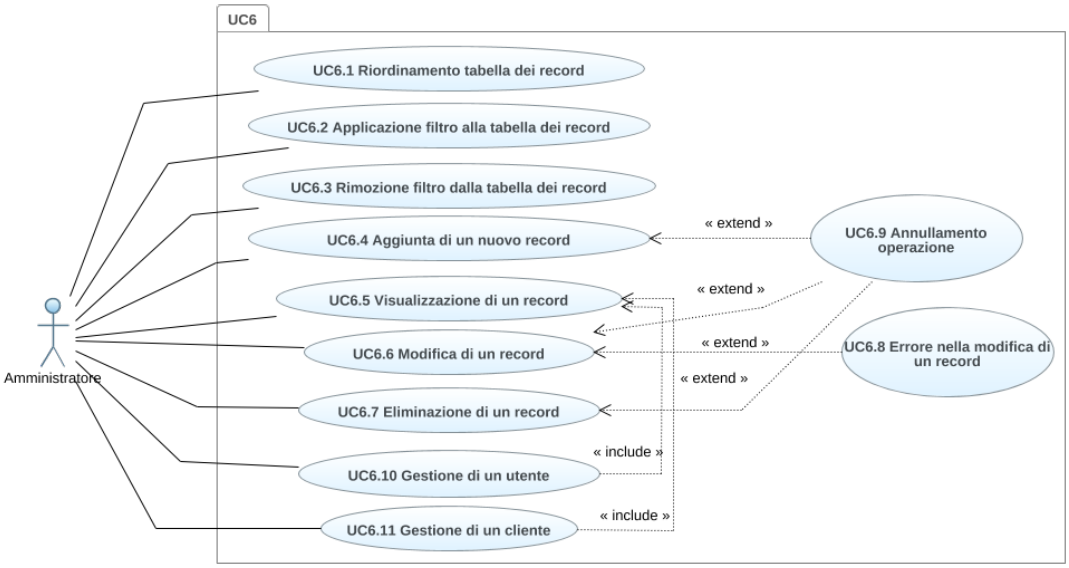

Figura 3.5: Diagramma UML per UC6

Attori: Amministratore.
Scopo e descrizione: L'attore vuole gestire una particolare categoria di record.
Precondizioni: L'attore è nella *dashboard* del pannello di amministrazione.
Scenario principale:

- L'attore seleziona una scheda associato alla categoria desiderata e visualizza la tabella dei record.
- L'attore modifica l'ordine della tabella dei record (UC6.1);
- L'attore applica un filtro alla tabella dei record (UC6.2);
- L'attore rimuove un filtro dalla tabella dei record (UC6.3);
- L'attore aggiunge un nuovo record (UC6.4);
- L'attore visualizza un record particolare (UC6.5);
- L'attore modifica un record particolare (UC6.6);
- L'attore elimina un record particolare (UC6.7);

- L'attore gestisce un utente (UC6.10).
- L'attore gestisce un cliente (UC6.11).

Postcondizioni: L'attore ha eseguito tutte le operazioni di gestione della categoria di record selezionata con successo.

3.1.7.2 UC7: Visualizzazione dashboard

Attori: Amministratore.
Scopo e descrizione: L'attore vuole visualizzare la *dashboard* del pannello di amministrazione.
Precondizioni: L'attore sta visualizzando una pagina diversa dalla *dashboard*.
Scenario principale: L'attore seleziona la scheda "Dashboard".
Postcondizioni: L'attore viene reindirizzato alla pagina della *dashboard*.

3.1.7.3 UC8: Gestione profilo

Attori: Amministratore.
Scopo e descrizione: L'attore vuole gestire il proprio profilo utente.
Precondizioni: L'attore ha accesso al pannello di amministrazione.
Scenario principale:

- L'attore clicca sull'icona del profilo.
- L'attore visualizza il record relativo al proprio profilo (UC6.5)

Postcondizioni: L'attore ha gestito correttamente il proprio profilo.

3.1.7.4 UC9: Disconnessione

Attori: Amministratore.
Scopo e descrizione: L'attore vuole disconnettersi dal pannello di amministrazione.
Precondizioni: L'attore ha accesso al pannello di amministrazione.
Scenario principale: L'attore clicca sull'icona di disconnessione.
Postcondizioni: L'attore viene correttamente disconnesso dal pannello di amministrazione.

3.1.8 UCG2: Operazioni per utente "pharmacy"

Questo caso d'uso generale riassume le operazioni disponibili al *front end* quando è attiva una sessione per un utente con ruolo di "pharmacy". Come anticipato, le operazioni vengono considerate non dal punto di vista dell'utente finale della piattaforma, cioè il farmacista o cliente che si interfaccia con il *front end*, ma come richieste che l'istanza di *front end* può effettuare alle API del servizio *back end*. Non verranno, per questo motivo, quindi fatte distinzioni tra interazioni che verrebbero effettuate da un farmacista o da un cliente. Infine, va considerato implicito che ognuna delle seguenti operazioni è ristretta ai record del database resi visibili dalla sessione attiva (nella fattispecie i record associati alla farmacia a cui appartiene l'utente autenticato).

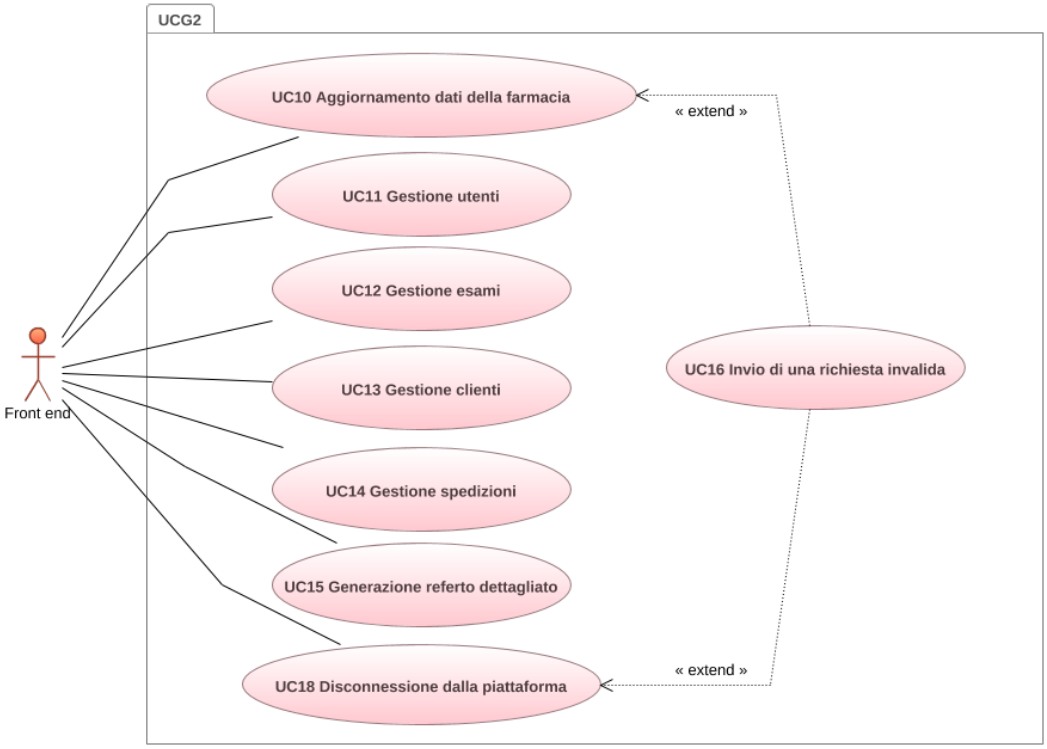

Figura 3.6: Diagramma UML per UCG2

Attori: Front end.
Scopo e descrizione: L'attore vuole eseguire operazioni permesse ad un utente con ruolo di "pharmacy".

Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore modifica i dati della farmacia (UC10);
- L'attore gestisce gli utenti con ruolo "pharmacy" (UC11);
- L'attore gestisce gli esami (UC12);
- L'attore gestisce i clienti (UC13);
- L'attore gestisce le spedizioni (UC14);
- L'attore visualizza il referto dettagliato per un esame (UC15);
- L'attore effettua la disconnessione dalla piattaforma (UC18).

Postcondizioni: L'attore ha correttamente portato a termine le operazioni desiderate.

3.1.8.1 UC10: Aggiornamento dati della farmacia

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare i dati della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia una richiesta di aggiornamento del record della farmacia con i nuovi dati, potenzialmente allegando un'immagine da sostituire al logo corrente.
Postcondizioni: Il record corrispondente alla farmacia viene aggiornato e l'attore ha ricevuto una risposta che conferma il successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

3.1.8.2 UC11: Gestione utenti

Attori: Front end.
Scopo e descrizione: L'attore vuole gestire gli altri utenti con ruolo "pharmacy" appartenenti alla farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

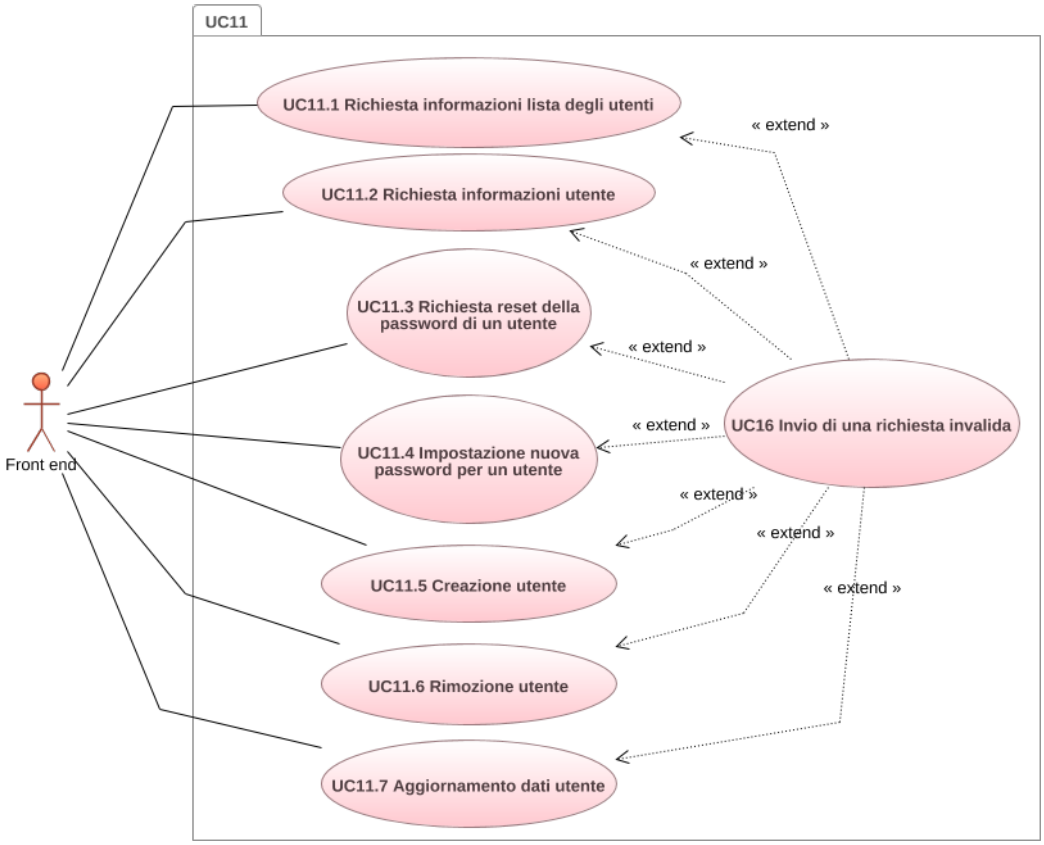

Figura 3.7: Diagramma UML per UC11

- L'attore richiede la lista degli utenti (UC11.1);
- L'attore richiede i dati di un utente (UC11.2);
- L'attore richiede il reset della password di un utente (UC11.3);
- L'attore imposta una nuova password per un utente (UC11.4);
- L'attore crea un utente (UC11.5);
- L'attore elimina un utente (UC11.6);
- L'attore aggiorna i dati di un utente (UC11.7).

Postcondizioni: L'attore ha portato a termine con successo le operazioni di gestione degli utenti.

3.1.8.3 UC12: Gestione esami

Figura 3.8: Diagramma UML per UC12

Attori: Front end.
Scopo e descrizione: L'attore vuole gestire gli altri utenti con ruolo "pharmacy" appartenenti alla farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore richiede lo storico degli esami (UC12.1);
- L'attore richiede lo storico degli esami in base allo stato di avanzamento (UC12.2);

- L'attore richiede i dati un esame (UC12.3);
- L'attore richiede i dati di esame in base al codice campione (UC12.4);
- L'attore crea un nuovo esame (UC12.5);
- L'attore rimuove un esame (UC12.6);
- L'attore aggiorna i dati di un esame (UC12.7);
- L'attore aggiorna le risposte al questionario di un'esame (UC12.8).

Postcondizioni: L'attore ha portato a termine con successo le operazioni di gestione degli utenti.

3.1.8.4 UC13: Gestione clienti

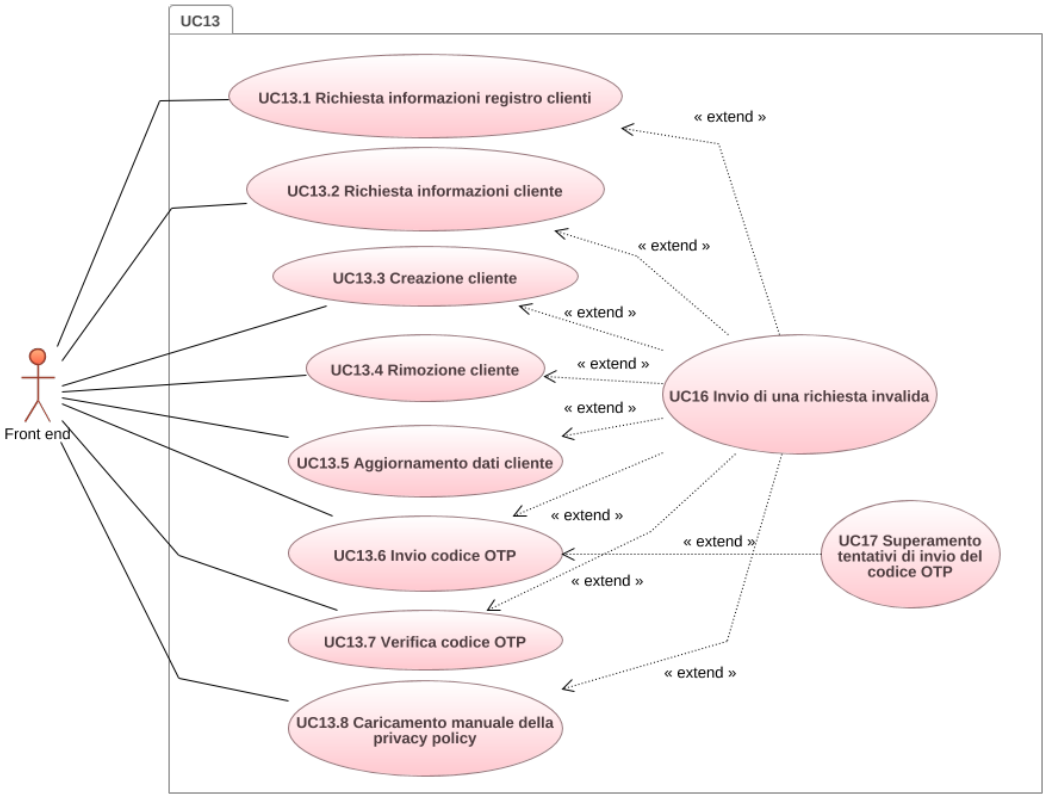

Figura 3.9: Diagramma UML per UC13

Attori: Front end.
Scopo e descrizione: L'attore vuole gestire i clienti appartenenti alla farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore richiede il registro dei clienti (UC13.1);
- L'attore richiede i dati di un cliente (UC13.2);
- L'attore crea un cliente (UC13.3);
- L'attore elimina un cliente (UC13.4);
- L'attore aggiorna i dati di un cliente (UC13.5);
- L'attore invia il codice per accettare la *privacy policy* ad un cliente (UC13.6);
- L'attore verifica l'accettazione della *privacy policy* per un cliente (UC13.7);
- L'attore effettua il caricamento di un modulo per il consenso alla *privacy policy* firmato da un cliente (UC13.8).

Postcondizioni: L'attore ha portato a termine con successo le operazioni di gestione dei clienti.

3.1.8.5 UC14: Gestione spedizioni

Attori: Front end.
Scopo e descrizione: L'attore vuole gestire le spedizioni per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore visualizza la lista delle spedizioni (UC14.1);
- L'attore visualizza una spedizione (UC14.2);
- L'attore crea una spedizione (UC14.3);
- L'attore elimina una spedizione (UC14.4);
- L'attore modifica i dati di una spedizione (UC14.5);

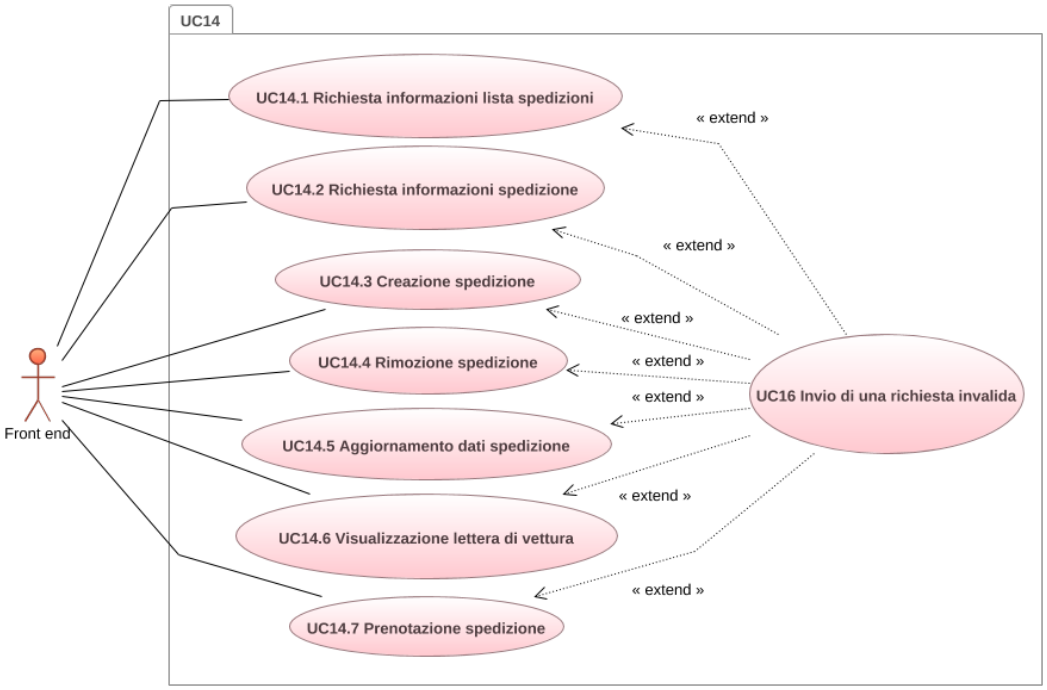

Figura 3.10: Diagramma UML per UC14

- L'attore visualizza la lettera di vettura per una spedizione (UC14.6);
- L'attore prenota il ritiro di una spedizione (UC14.7).

Postcondizioni: L'attore ha portato a termine con successo le operazioni di gestione delle spedizioni.

3.1.8.6 UC15: Generazione referto dettagliato

Attori: Front end.
Scopo e descrizione: L'attore vuole visualizzare il referto di un esame, completo delle informazioni specifiche risultate dall'analisi e delle informazioni personali del cliente.
Precondizioni: L'attore è collegato al sistema ed l'esame desiderato è arrivato correttamente in stato **final**.
Scenario principale:

- L'attore richiede la visualizzazione del referto, fornendo il codice identificativo interno dell'esame cui è associato;

- Il sistema processa il file PDF del referto inviato dal laboratorio aggiungendo la diagnosi del gastroenterologo, la tabella della dieta e le generalità del paziente;
- L'attore riceve dal sistema il referto generato.

Postcondizioni: L'attore ha visualizzato correttamente il referto specifico.
Estensioni: L'attore invia una richiesta non valida (UC16).

3.1.8.7 UC16: Invio di una richiesta invalida

Attori: Front end.
Scopo e descrizione: L'attore vuole inviare una richiesta al sistema.
Precondizioni: L'attore è collegato al sistema e possiede una richiesta contenente dati non validi.
Scenario principale:

- L'attore invia la richiesta con dati non validi;
- L'attore riceve un messaggio di errore che descrive la violazione commessa.

Postcondizioni: La richiesta dell'attore viene rifiutata e l'attore ha ricevuto un messaggio di errore appropriato.

3.1.8.8 UC17: Superamento tentativi di invio del codice OTP

Attori: Front end.
Scopo e descrizione: L'attore vuole inviare il codice per l'accettazione della privacy policy ad un cliente della farmacia corrente.
Precondizioni: Sono già stati effettuati 3 tentativi nelle ultime 24 ore.
Scenario principale:

- L'attore richiede l'invio del codice;
- L'attore riceve un messaggio di errore che lo informa del superamento del numero massimo di tentativi per la giornata.

Postcondizioni: La richiesta dell'attore viene rifiutata e l'attore ha ricevuto un messaggio di errore appropriato.

3.1.8.9 UC18: Disconnessione dalla piattaforma

Attori: Front end.
Scopo e descrizione: L'attore vuole disconnettersi dalla piattaforma.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva.
Scenario principale:

- L'attore richiede la disconnessione dalla piattaforma;
- Il sistema invalida i parametri della sessione corrente.

Postcondizioni: L'attore viene disconnesso correttamente e il sistema non accetta più richieste con i parametri della sessione invalidata.

3.1.9 UCG3: Operazioni per utente "gastroenterologist"

Questo caso d'uso generale riassume le operazioni disponibili al *front end* quando è attiva una sessione per un utente con ruolo di "gastroenterologist". Valgono le stesse considerazioni fatte per UCG2, con l'eccezione dei record disponibili: nella descrizione dei seguenti casi d'uso le operazioni non sono ristrette ai record associati ad una farmacia, ma escludono rigorosamente i dati personali dei clienti.
Attori: Front end.
Scopo e descrizione: L'attore vuole eseguire operazioni permesse ad un utente con ruolo di "gastroenterologist".
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "gastroenterologist".
Scenario principale:

- L'attore richiede la lista di esami in attesa di diagnosi (UC19);
- L'attore richiede i dati di un esame in attesa di diagnosi (UC20);
- L'attore inserisce una diagnosi per un esame (UC21);
- L'attore inserisce la tabella della dieta per un esame (UC22);
- L'attore visualizza il referto dettagliato anonimo per un esame (UC23);
- L'attore effettua la disconnessione dalla piattaforma (UC18).

Postcondizioni: L'attore ha correttamente portato a termine le operazioni desiderate.

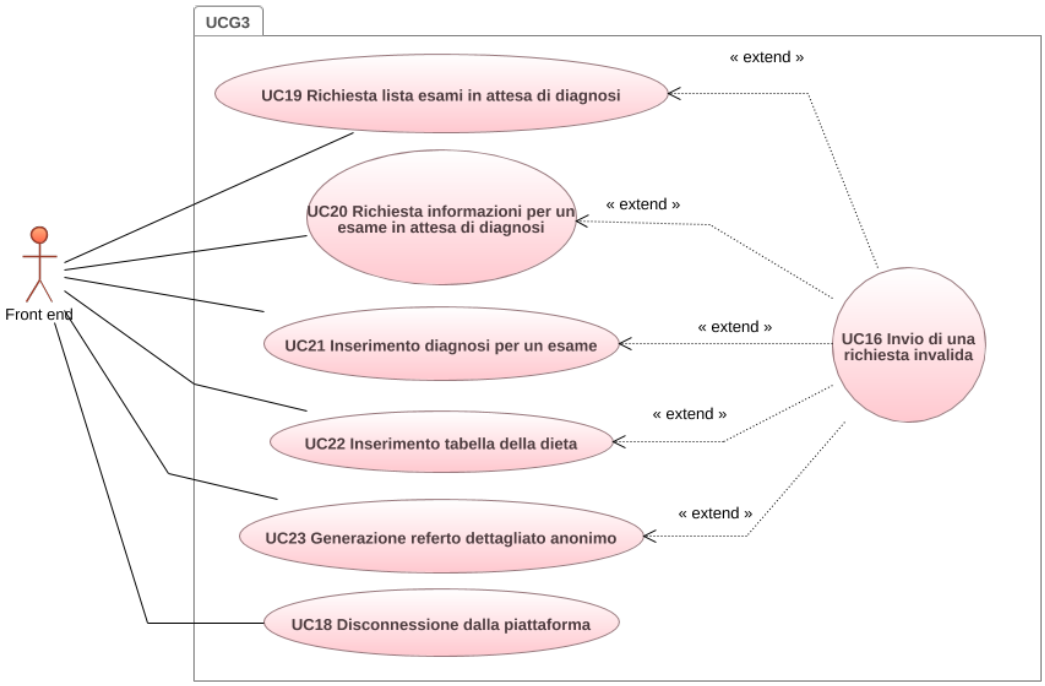

Figura 3.11: Diagramma UML per UCG3

3.1.9.1 UC19: Richiesta lista esami in attesa di diagnosi

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere la lista degli esami per cui è disponibile il referto delle analisi e sono quindi in attesa di diagnosi.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "gastroenterologist".
Scenario principale: L'attore invia una richiesta per ottenere la lista di esami desiderata.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

3.1.9.2 UC20: Richiesta informazioni per un esame in attesa di diagnosi

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un esame per cui è disponibile il referto delle analisi ed è quindi in attesa di diagnosi.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente

con ruolo “gastroenterologist”.
Scenario principale: L’attore invia una richiesta per ottenere l’esame desiderato, fornendone il codice identificativo interno .
Postcondizioni: L’attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.9.3 UC21: Inserimento diagnosi per un esame

Attori: Front end.
Scopo e descrizione: L’attore vuole inserire una diagnosi per un esame.
Precondizioni: L’attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo “gastroenterologist”, l’esame rientra tra quelli visibili.
Scenario principale: L’attore invia una richiesta di inserimento della diagnosi per l’esame desiderato contenente le informazioni generali e specifiche della diagnosi, fornendo il codice identificativo interno dell’esame.
Postcondizioni: Il sistema ha registrato correttamente la diagnosi e l’attore ha ricevuto una risposta che conferma il successo dell’operazione.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.9.4 UC22: Inserimento tabella della dieta

Attori: Front end.
Scopo e descrizione: L’attore vuole inserire la tabella della dieta associata ad un esame.
Precondizioni: L’attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo “gastroenterologist”, l’esame rientra tra quelli visibili.
Scenario principale: L’attore invia una richiesta di inserimento della tabella della dieta per l’esame desiderato, fornendo il codice identificativo interno dell’esame.
Postcondizioni: Il sistema ha registrato correttamente la tabella della dieta e l’attore ha ricevuto una risposta che conferma il successo dell’operazione.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.9.5 UC23: Generazione referto dettagliato anonimo

Attori: Front end.
Scopo e descrizione: L’attore vuole visualizzare il referto di un esame, completo delle informazioni specifiche risultate dall’analisi.

Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "gastroenterologist" o "bmr", l'esame rientra tra quelli visibili.
Scenario principale:

- L'attore richiede la visualizzazione del referto, fornendo il codice identificativo interno dell'esame cui è associato;
- Il sistema processa il file PDF del referto inviato dal laboratorio aggiungendo la diagnosi del gastroenterologo e la tabella della dieta;
- L'attore riceve dal sistema il referto generato.

Postcondizioni: L'attore ha visualizzato correttamente il referto specifico.
Estensioni: L'attore invia una richiesta non valida (UC16).

3.1.10 UCG4: Operazioni per utente "bmr"

Questo caso d'uso generale riassume le operazioni disponibili al *front end* quando è attiva una sessione per un utente con ruolo di "bmr" (supervisore del laboratorio). Valgono le stesse considerazioni fatte per UCG3.
Attori: Front end.
Scopo e descrizione: L'attore vuole eseguire operazioni permesse ad un utente con ruolo di "bmr".
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "bmr".
Scenario principale:

- L'attore richiede la lista di esami per cui è stata emessa una diagnosi (UC24);
- L'attore richiede i dati di un esame per cui è stata emessa una diagnosi (UC25);
- L'attore aggiorna una diagnosi per un esame (UC26);
- L'attore aggiorna la tabella della dieta per un esame (UC27);
- L'attore visualizza il referto dettagliato anonimo per un esame (UC23);
- L'attore effettua la disconnessione dalla piattaforma (UC18).

Postcondizioni: L'attore ha correttamente portato a termine le operazioni desiderate.

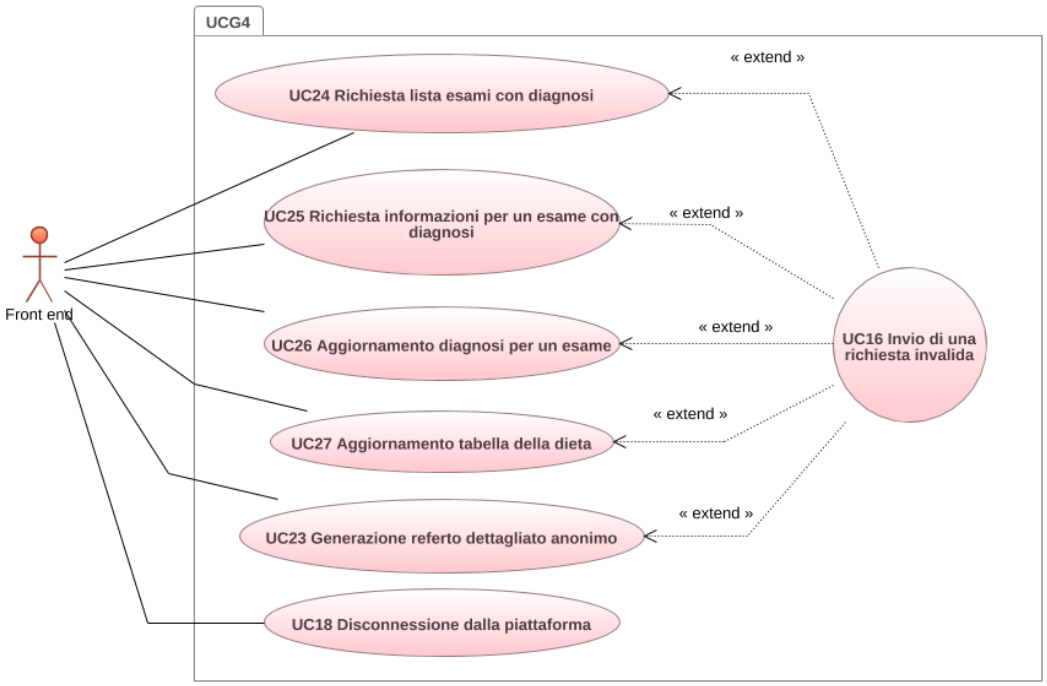

Figura 3.12: Diagramma UML per UCG4

3.1.10.1 UC24: Richiesta lista esami con diagnosi

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere la lista degli esami per cui è stata emessa una diagnosi.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "bmr".
Scenario principale: L'attore invia una richiesta per ottenere la lista di esami desiderata.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

3.1.10.2 UC25: Richiesta informazioni esame con diagnosi

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un esame per cui è stata emessa una diagnosi.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente

con ruolo “gastroenterologist”.
Scenario principale: L’attore invia una richiesta per ottenere l’esame desiderato, fornendone il codice identificativo interno .
Postcondizioni: L’attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.10.3 UC26: Aggiornamento diagnosi per un esame

Attori: Front end.
Scopo e descrizione: L’attore vuole modificare la diagnosi emessa per un esame.
Precondizioni: L’attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo “bmr”, l’esame rientra tra quelli visibili.
Scenario principale: L’attore invia una richiesta di modifica della diagnosi per l’esame desiderato contenente le nuove informazioni generali e specifiche della diagnosi, fornendo il codice identificativo interno dell’esame.
Postcondizioni: Il sistema ha aggiornato correttamente la diagnosi e l’attore ha ricevuto una risposta che conferma il successo dell’operazione.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.10.4 UC27: Aggiornamento tabella della dieta

Attori: Front end.
Scopo e descrizione: L’attore vuole aggiornare la tabella della dieta associata ad un esame.
Precondizioni: L’attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo “bmr”, l’esame rientra tra quelli visibili.
Scenario principale: L’attore invia una richiesta di modifica della tabella della dieta per l’esame desiderato, fornendo il codice identificativo interno dell’esame.
Postcondizioni: Il sistema ha aggiornato correttamente la tabella della dieta e l’attore ha ricevuto una risposta che conferma il successo dell’operazione.
Estensioni: L’attore invia una richiesta non valida (UC16).

3.1.11 UCG5: Operazioni per il gestionale del laboratorio

Questo caso d’uso generale riassume le operazioni disponibili al software gestionale del laboratorio attraverso gli appositi *endpoint* HTTP. Si prevede che le richieste HTTP contengano,

nella sezione *header*, le credenziali necessarie per l'autenticazione tramite HMAC.

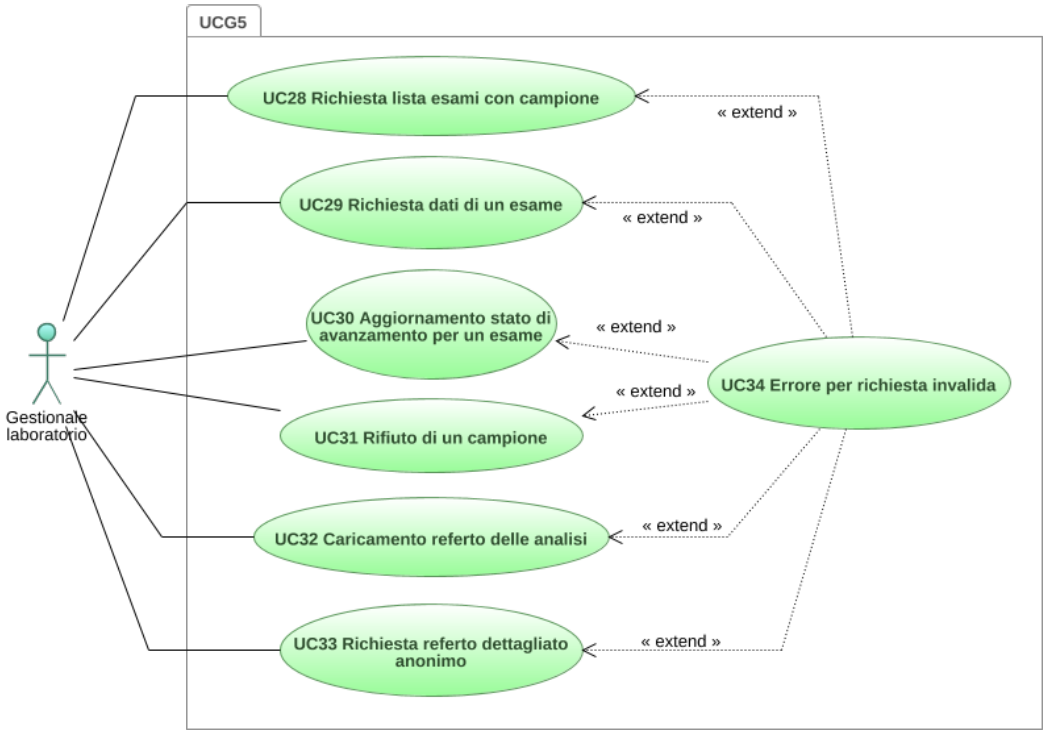

Figura 3.13: Diagramma UML per UCG5

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole eseguire operazioni rese disponibili dagli *endpoint* esposti.
Precondizioni: L'attore ha accesso agli *endpoint* forniti dal sistema e possiede le credenziali necessarie per l'autenticazione HMAC.
Scenario principale:

- L'attore richiede la lista di esami il cui campione è stato raccolto (UC28);
- L'attore richiede i dati di un esame (UC29);
- L'attore aggiorna lo stato di avanzamento di un esame (UC30);
- L'attore rifiuta il campione per un esame (UC31);
- L'attore carica il referto delle analisi (UC32).

- L'attore richiede il referto dettagliato anonimo per un esame (UC33)

Postcondizioni: L'attore ha correttamente portato a termine le operazioni desiderate.

3.1.ii.1 UC28: Richiesta lista esami con campione

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole ottenere la lista degli esami il cui campione è stato raccolto, compresi quelli il cui iter di avanzamento è stato completato.
Precondizioni: L'attore ha accesso all'*endpoint* corrispondente e ha autenticato correttamente la richiesta.
Scenario principale: L'attore invia la richiesta HTTP di sincronizzazione della lista degli esami all'*endpoint* appropriato.
Postcondizioni: L'attore ha ricevuto una risposta contenente i dati richiesti in formato JSON.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.ii.2 UC29: Richiesta dati di un esame

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole ottenere i dati di un esame.
Precondizioni: L'attore ha accesso all'*endpoint* corrispondente e ha autenticato correttamente la richiesta.
Scenario principale: L'attore invia la richiesta HTTP di ricezione dei dati dell'esame all'*endpoint* appropriato.
Postcondizioni: L'attore ha ricevuto una risposta contenente i dati richiesti in formato JSON.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.ii.3 UC30: Aggiornamento stato di avanzamento di un esame

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole aggiornare lo stato di avanzamento di un esame ad uno di quello disponibili.

Precondizioni: L'attore ha accesso all'*endpoint* corrispondente e ha autenticato correttamente la richiesta.
Scenario principale: L'attore invia la richiesta HTTP corretta per il nuovo stato dell'esame all'*endpoint* appropriato, specificando il codice campione a cui l'esame è associato.
Postcondizioni: L'attore ha ricevuto una risposta che conferma il successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.II.4 UC31: Rifiuto di un campione

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole segnalare che il campione per un determinato esame è stato rifiutato.
Precondizioni: L'attore ha accesso all'*endpoint* corrispondente e ha autenticato correttamente la richiesta.
Scenario principale:

- L'attore invia la richiesta HTTP per rifiutare del campione all'*endpoint* appropriato, specificando il codice campione a cui l'esame è associato e inserendo nel *body* la ragione del rifiuto.
- Il sistema effettua il *parsing* della risposta per ottenere la ragione del rifiuto e aggiorna l'esame desiderato.

Postcondizioni: Il sistema ha aggiornato correttamente l'esame e l'attore ha ricevuto una risposta che conferma il successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.II.5 UC32: Caricamento referto delle analisi

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole caricare il referto delle analisi per un esame, assieme ad eventuali appunti sul referto.
Precondizioni: L'attore ha accesso all'*endpoint* corrispondente, ha autenticato correttamente la richiesta ed è disponibile un referto da caricare.
Scenario principale:

- L'attore invia la richiesta HTTP di conclusione delle analisi all'*endpoint* appropriato, allegando il referto ed eventuali note alle analisi e specificando il codice campione a cui l'esame è associato.
- Il sistema effettua il *parsing* della risposta e aggiorna il record dell'esame corrispondente aggiungendo il referto ed eventuali note.

Postcondizioni: Il sistema ha ricevuto il referto, ha aggiornato correttamente l'esame e l'attore ha ricevuto una risposta che conferma il successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.II.6 UC33: Richiesta referto dettagliato anonimo

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore vuole ricevere il referto dettagliato con la diagnosi del gastroenterologo per un esame.
Precondizioni: L'attore ha accesso all'*endpoint* corrispondente, ha autenticato correttamente la richiesta ed è disponibile un referto da visualizzare.
Scenario principale:

- L'attore invia la richiesta HTTP di visualizzazione del referto all'*endpoint* appropriato.
- Il sistema elabora il referto dettagliato e e lo restituisce all'attore.

Postcondizioni: L'attore ha ricevuto il referto dettagliato.
Estensioni: L'attore invia una richiesta non valida e ottiene una risposta che segnala l'errore (UC34).

3.1.II.7 UC34: Errore per richiesta invalida

Attori: Gestionale laboratorio.
Scopo e descrizione: L'attore invia una richiesta contenente dati non validi.
Precondizioni: L'attore ha accesso all'*endpoint* desiderato e ha autenticato correttamente la richiesta.
Scenario principale: L'attore invia la richiesta HTTP contenente dati non validi all'*endpoint* desiderato. **Postcondizioni:** L'attore ha visualizzato un risposta di errore che descrive la violazione commessa.

3.2 Tracciamento dei requisiti

Parte dell'analisi dei requisiti è stato anche, appunto, l'individuare i requisiti del prodotto con granularità più fine rispetto agli obiettivi generali pianificati; dopodiché è stato possibile tracciarne i rapporti con i casi d'uso. Di seguito vengono riportati i requisiti individuati in forma tabulare; per il tracciamento con i relativi casi d'uso, consultare l'appendice B.

3.2.1 Notazione

Si farà riferimento ai requisiti secondo la seguente notazione:

R{tipo}{identificativo}

- **R**: indica che si tratta di un requisito;
- **tipo**: indica la categoria del requisito, può essere F (funzionale), Q (qualitativo) o V (di vincolo);
- **identificativo**: numero progressivo che serve a distinguere i vari sotto-requisiti.

3.2.2 Requisiti funzionali

Sigla	Descrizione
RF1	L'utente non autenticato deve poter accedere al pannello di amministrazione.
RF1.1	L'utente non autenticato deve poter inserire la propria mail.
RF1.2	L'utente non autenticato deve poter inserire la propria password.
RF1.3	L'utente non autenticato deve visualizzare un messaggio di errore se inserisce le credenziali errate.
RF2	Il *front end* deve poter autenticare un utente autorizzato ottenendo i parametri di una sessione attiva.
RF2.1	Il *front end* deve ricevere un messaggio di errore in seguito ad un tentativo di autenticazione con credenziali errate.
RF3	Il *front end* deve poter inviare una richiesta di reset della password per un utente autorizzato.
RF4	L'amministratore deve poter gestire una categoria di record.

RF4.1	L'amministratore deve poter modificare l'ordine della tabella dei record.
RF4.2	L'amministratore deve poter applicare un filtro alla tabella dei record.
RF4.3	L'amministratore deve poter rimuovere un filtro dalla tabella dei record.
RF4.4	L'amministratore deve poter creare un nuovo record.
RF4.5	L'amministratore deve poter visualizzare un record particolare.
RF4.6	L'amministratore deve poter modificare un record particolare.
RF4.7	L'amministratore deve poter rimuovere un record particolare.
RF4.8	L'amministratore deve poter gestire un utente.
RF4.8.1	L'amministratore deve poter generare una nuova password per un utente.
RF4.9	L'amministratore deve poter gestire un cliente.
RF4.9.1	L'amministratore deve poter scaricare la *privacy policy* di un cliente.
RF4.10	L'amministratore deve poter annullare una operazione.
RF4.11	L'amministratore deve visualizzare un messaggio di errore se modifica un record inserendo valori invalidi.
RF5	L'amministratore deve poter visualizzare la *dashboard*.
RF6	L'amministratore deve poter gestire il proprio profilo utente.
RF7	L'amministratore deve potersi disconnettere dal pannello di amministrazione.
RF8	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter modificare i dati della farmacia.
RF9	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter gestire gli utenti della stessa farmacia.
RF9.1	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter richiedere la lista degli utenti della stessa farmacia.
RF9.2	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter visualizzare un utente della stessa farmacia.
RF9.3	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter richiedere il reset della password per un utente della stessa farmacia.

RF9.4	Il *front end* (in possesso del *token* di reset) deve poter impostare una nuova password per un utente della stessa farmacia.
RF9.5	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter creare un nuovo utente della stessa farmacia.
RF9.6	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter eliminare un utente farmacia.
RF9.7	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter aggiornare un utente della stessa farmacia.
RF10	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter gestire gli esami della farmacia.
RF10.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere lo storico degli esami della farmacia.
RF10.1.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere la lista degli esami in base allo stato di avanzamento della farmacia.
RF10.2	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere dati di un esame della farmacia.
RF10.2.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere dati di un esame della farmacia in base al codice campione.
RF10.3	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter creare un nuovo esame.
RF10.4	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter rimuovere un esame.
RF10.5	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter aggiornare un esame.
RF10.5.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter aggiornare le risposte al questionario di un esame.
RF11	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter gestire i clienti della farmacia.
RF11.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere il registro dei clienti della farmacia.

RF11.2	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere i dati di un cliente della farmacia.
RF11.3	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter creare un nuovo cliente.
RF11.4	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter rimuovere un cliente.
RF11.5	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter aggiornare un cliente.
RF11.6	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter inviare un codice OTP per accettare la *privacy policy* ad un cliente.
RF11.6.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve visualizzare un messaggio di errore se supera i tentativi giornalieri di invio di un codice OTP.
RF11.7	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter verificare un codice OTP per l’accettazione della *privacy policy* da parte un cliente.
RF11.8	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter caricare una copia della *privacy policy* firmata da un cliente.
RF12	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter gestire le spedizioni della farmacia.
RF12.1	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere la lista delle spedizioni della farmacia.
RF12.2	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter richiedere i dati di una spedizione della farmacia.
RF12.3	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter creare una nuova spedizione.
RF12.4	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter rimuovere una spedizione.
RF12.5	Il *front end* (utente autenticato con ruolo “pharmacy”) deve poter aggiornare una spedizione.

RF12.6	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter visualizzare la lettera di vettura per una spedizione.
RF12.7	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter prenotare una spedizione.
RF13	Il *front end* (utente autenticato con ruolo "pharmacy") deve poter generare un referto dettagliato per un cliente.
RF14	Il *front end* (utente autenticato con ruolo qualsiasi) deve visualizzare un messaggio di errore quando viene inviata una richiesta invalida.
RF15	Il *front end* (utente autenticato con ruolo qualsiasi) potersi disconnettere dalla piattaforma.
RF16	Il *front end* (utente autenticato con ruolo "gastroenterologist") deve poter richiedere la lista degli esami in attesa di diagnosi.
RF17	Il *front end* (utente autenticato con ruolo "gastroenterologist") deve poter richiedere i dati di un esame in attesa di diagnosi.
RF18	Il *front end* (utente autenticato con ruolo "gastroenterologist") deve poter inserire una diagnosi per un esame.
RF19	Il *front end* (utente autenticato con ruolo "gastroenterologist") deve poter inserire la tabella della dieta per un esame.
RF20	Il *front end* (utente autenticato con ruolo "gastroenterologist" o "bmr") deve poter richiedere il referto dettagliato anonimo per un esame.
RF21	Il *front end* (utente autenticato con ruolo "bmr") deve poter richiedere la lista degli esami per cui è stata emessa una diagnosi.
RF22	Il *front end* (utente autenticato con ruolo "bmr") deve poter richiedere i dati di un esame per cui è stata emessa una diagnosi.
RF23	Il *front end* (utente autenticato con ruolo "bmr") deve poter aggiornare la diagnosi per un esame.
RF24	Il *front end* (utente autenticato con ruolo "bmr") deve poter aggiornare la tabella della dieta per un esame.

RF25	Il laboratorio deve poter richiedere la lista degli esami il cui campione è stato raccolto.
RF26	Il laboratorio deve poter richiedere i dati di un esame.
RF27	Il laboratorio deve poter aggiornare lo stato di avanzamento di un esame.
RF28	Il laboratorio deve poter rifiutare il campione associato ad un esame.
RF29	Il laboratorio deve poter caricare il referto delle analisi per un esame.
RF30	Il laboratorio deve poter richiedere il referto dettagliato per un esame.
RF31	Il laboratorio deve poter visualizzare una risposta di errore se effettua una richiesta non valida.

Tabella 3.1: Tracciamento dei requisiti funzionali.

3.2.3 Requisiti qualitativi

Sigla	Descrizione
RQ1	Il prodotto deve essere codificato secondo le norme esposte dalla guida stilistica della community di Ruby on Rails.
RQ2	Il prodotto deve essere versionato seguendo il paradigma *git-flow*.
RQ3	Il prodotto deve essere verificato e validato prima del rilascio di una nuova *feature*.
RQ4	Deve essere sviluppata una suite di test per il prodotto.
RQ5	Deve essere stesa una documentazione sufficiente per il prodotto.
RQ5.1	Deve essere redatta una precisa documentazione delle API di interazione con il sistema gestionale del laboratorio.
RQ6	Il prodotto deve essere progettato secondo le norme di Ruby on Rails.
RQ6.1	La progettazione del prodotto deve rispettare il principio DRY.
RQ6.2	La progettazione del prodotto deve rispettare il principio CoC.

Tabella 3.2: Tracciamento dei requisiti qualitativi.

3.2.4 Requisiti di vincolo

Sigla	Descrizione
RV1	Il prodotto deve essere sviluppato con il *framework* Ruby on Rails 5.
RV1.1	Il prodotto deve essere sviluppato con il linguaggio Ruby ed essere compatibile con la versione 2.5.5.
RV2	Il prodotto deve essere compatibile con il *back end* esistente.
RV3	Il prodotto deve mettere interfacciarsi con il *front end* esponendo una appropriata API GraphQL.
RV3.1	L'autenticazione delle richieste GraphQL deve avvenire tramite la gemma Devise.
RV4	Il prodotto deve mettere a disposizione una API REST per l'interazione con il software gestionale del laboratorio.
RV4.1	L'autenticazione delle richieste HTTP deve avvenire mediante lo standard HMAC.

Tabella 3.3: Tracciamento dei requisiti di vincolo.

I'll be more enthusiastic about encouraging thinking outside the box when there's evidence of any thinking going on inside it.

Terry Pratchett

4 Progettazione

La natura di questo prodotto come parte di un ecosistema esistente e già affermato ha avuto un duplice effetto sulla fase di progettazione. Da un lato l'ha resa più semplice, potendo appoggiarsi a codice preesistente e sfruttando meccanismi e servizi già pronti all"uso come le funzionalità di autenticazione (basate sulla gemma Devise) e il *mailer* configurato; dall'altro, tuttavia, ha fatto emergere diverse sfide nell'integrazione del codice necessario per implementare nuove funzionalità con il *code base* presente. L'applicazione condivide infatti buona parte della struttura di base con un altro applicativo dell'ecosistema, ed è quindi stata progettata per essere implementata in modo compatibile a quest'ultimo - un esempio è l'aggiunta di domande a risposta multipla ai questionari, che ha comportato un notevole *refactoring* del codice dedicato. In questa sezione verrà presentata l'architettura generale dell'applicazione, inclusi i paradigmi su cui si fonda, e verranno descritte le sue parti fondamentali. Ove queste parti si sovrappongano all'architettura preesistente, verranno indicate le eventuali aggiunte o modifiche apportate nello corso della progettazione.

4.1 Progettare con Rails

Come anticipato nella sezione 2.2.3, l'utilizzo di Ruby on Rails comporta il seguire certi paradigmi nella scelta dell'architettura e nella progettazione di un'applicazione. Progettare con Rails è sicuramente stata un'esperienza molto diversa rispetto all'utilizzo di altri framework,

per via della sua natura altamente integrata che guida la progettazione verso un certo stile: le applicazioni sviluppate con Rails sono fortemente orientate ad uno stile architetturale "*REST-like*", in cui la *business logic* e il database sono fortemente accoppiati in *models* che costituiscono le risorse; tali risorse sono quindi gestite attraverso dei *controllers*, le cui operazioni sono rese accessibili attraverso *endpoint* specificati nel file di *routing*, e rese visibili grazie alle *views* associate ai *controllers*.

Altre peculiarità di Rails risiede nel fatto che, per lo stesso motivo, il classico paradigma di ereditarietà della programmazione ad oggetti non risulta ottimale per l'implementazione dei *models* che sono al cuore della *business logic* dell'applicativo: essendo tali classi convenzionalmente associate a tabelle del database, la loro progettazione deve tenere conto dell'aumento di complessità del database che sarebbe dovuto all'applicazione di classici *design pattern*. Spesso quindi è preferibile aggiungere direttamente campi (e relativi metodi) ad un *model* ed usare questi per differenziarne il comportamento in situazioni diverse. È il caso, come verrà spiegato più avanti, dei modelli che rappresentano gli esami: creare un nuovo modello per supportare gli esami in "Biota", separato da quello preesistente, avrebbe aumentato inutilmente la complessità del database (in quanto la maggior parte dei campi sono condivisi, risultando in due tabelle quasi identiche); avrebbe inoltre comportato la duplicazione delle query GraphQL necessarie. Sono invece stati aggiunti i campi necessari al modello esistente, grazie alla presenza di un campo `service` che agisce da discriminante sull'applicazione cui appartiene un determinato esame.

4.2 Architettura

L'architettura di "Biota" si è necessariamente dovuta conformare a quella della piattaforma preesistente di cui fa parte, adottando uno stile *REST-like* e implementando un pattern MVC. Come menzionato nella sezione 2.1.1, l'applicazione è articolata in un *back end* e un *front end* sviluppati separatamente, in accordo allo stile REST che prevede la separazione di interfaccia utente e risorse. Il *back end* è stato progettato implementando un pattern MVC con uno stile architetturale REST; l'interazione tra *back end* e *front end* è permessa dall'utilizzo di API GraphQL attraverso cui il *back end* espone varie operazioni di visualizzazione, creazione, modifica ed eliminazione delle risorse. Sebbene normalmente il paradigma REST si basi direttamente sul protocollo HTTP, l'utilizzo di GraphQL permette la progettazione di una API più versatile facilitando il reperimento di informazioni. Viene inoltre resa disponibile una classica API HTTP per l'interazione con il software gestionale del laboratorio di

analisi. Entrambe le API sono progettate in modo da garantire la sicurezza, richiedendo che le richieste siano autenticate con diverse modalità.

La gestione manuale delle risorse è possibile attraverso un pannello di amministrazione dedicato, anch'esso realizzato secondo uno stile *REST-like*. Una piccola variazione rispetto a questo stile è rappresentata dalle interazioni con il servizio esterno Amazon SNS e le API del corriere SDA, che sono gestite da servizi dedicati in forma di *client* ad-hoc istanziati ed eseguiti dal *back end*, prendendo spunto dallo stile architetturale a micro-servizi.

4.2.1 Representational State Transfer

Il paradigma architetturale principale, già menzionato più volte, è REST, acronimo di *Representational State Transfer*; esso è uno degli stili architetturali più utilizzati nella progettazione di servizi Web grazie alla caratteristica di snellire lo scambio di informazioni su cui essi si basano.

Un sistema RESTful si compone di due parti: un **client** che richiede delle risorse e un **server** che possiede e gestisce tali risorse. Il client ed il server comunicano attraverso una apposita interfaccia detta API che implementa un protocollo di comunicazione tra le due entità e permette l'invio di richieste da parte del client, e l'invio di risorse da parte del server.

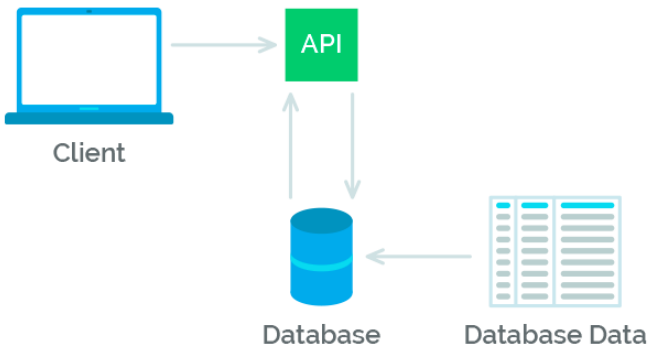

Figura 4.1: Rappresentazione semplificata dello stile REST.

L'elemento fondamentale dello stile REST sono, appunto, le risorse: unità di informazione basilari identificate univocamente da un URL che assomigliano, in un certo senso, ai classici oggetti alla base dell'*object-oriented programming*. Le risorse risiedono sul server e sono soggette a quattro tipi di operazioni, note come *CRUD* (*Create, Read, Update, Delete*):

- **Create**: crea una nuova risorsa;
- **Read**: rende disponibile una rappresentazione della risorsa;
- **Update**: modifica il valore della risorsa;
- **Delete**: rende la risorsa inaccessibile.

L'accesso alle risorse avviene esclusivamente attraverso queste operazioni, che possono essere ridefinite ed adattate secondo le necessità del client: ad esempio, diverse implementazioni di *read* possono restituire diverse rappresentazioni della stessa risorsa.

Nell'applicazione le risorse sono costituite da record del database descritte da classi dette *models*, che ne danno una rappresentazione interna e contengono la relativa *business logic* in forma di metodi.

4.2.1.1 GraphQL e HTTP

La principale differenza che l'applicazione presenta rispetto ai classici sistemi REST è l'utilizzo di GraphQL anziché HTTP per l'implementazione dell'API di comunicazione con in *front end*. Il protocollo HTTP, utilizzato come standard nella progettazione di servizi Web con architettura REST, prevede 4 metodi per il trasferimento dei dati tra client e server che corrispondono alle 4 operazioni CRUD: La rappresentazione delle risorse avviene nor-

Metodo	**Operazione**
POST	Create
GET	Read
PUT	Update
DELETE	Delete

Tabella 4.1: Corrispondenza tra metodi HTTP e operazioni CRUD.

malmente attraverso lo standard JSON come parte del *body* di una richiesta o risposta. Il client invia una richiesta al server utilizzando uno di questi metodi a un *endpoint* esposto dal server, che interpreta la richiesta e risponde di conseguenza; la struttura dei dati restituiti da ogni endpoint è fissata (la rappresentazione della risorsa è legata alla richiesta), quindi rappresentazioni anche parzialmente diverse delle risorse richiederanno *endpoint* diversi.

In GraphQL viene definito uno **schema** di tipi *server-side*, che descrive le risorse disponibili e le operazioni che è possibile eseguire su di esse. Le operazione si dividono in due categorie:

queries, corrispondenti all'operazione *create*, e *mutations*, che raggruppano le operazioni di *create, update* e *delete*. Le richieste GraphQL vengono inviate tramite protocollo HTTP ad un *endpoint* apposito esposto dal server, e quindi vengono gestite dal componente *run-time*.

Nell'operazione di ottenimento dei dati il client può inviare *query* diverse allo stesso *endpoint*: in questo modo, con una *query* appropriata, il client può ottenere esattamente i dati necessari risolvendo i problemi di *overfetching* e *underfetching* caratteristici dello stile REST. GraphQL mantiene il concetto di risorsa, l'invio di richieste mediante il protocollo HTTP e l'utilizzo dello standard JSON; tuttavia separa la rappresentazione della risorsa dal metodo per ottenerla, permettendo al client di richiedere esattamente le informazioni necessarie.

Figura 4.2: Differenza tra API REST classiche e API GraphQL.

4.2.2 Implementazione dello stile REST

Lo stile architetturale appena descritto, che riguarda l'intera piattaforma, è implementato nel seguente modo:

- **Server**: Il server corrisponde al *back end* sviluppato, che gestisce le risorse e risponde alle richieste dei client. Le risorse sono i record salvati nel database PostgreSQL del server, rappresentati come *models*, classi di oggetti realizzate utilizzando il modulo **ActiveRecord** di Rails. ActiveRecord facilita la gestione *RESTful* delle risorse implementando metodi appositi per i *models* le corrispondenti operazioni CRUD.

- **API**: Il server mette a disposizione due API, una GraphQL e una HTTP. Le richieste provenienti dall'*endpoint* dedicato a GraphQL sono interpretate dal *runtime system* di GraphQL e gestite secondo il sistema di tipi definito; le richieste HTTP sono reindirizzate al corrispondente metodo del controller dedicato secondo quanto definito nel file di *routing*.

- **Client**:

1. Il *front end* dell'applicazione interagisce tramite l'API GraphQL per svolgere le operazioni necessarie all'utilizzo dell'applicazione; l'utilizzo di GraphQL permette di al server di fornire esattamente le informazioni richieste, qualità utile all'interfaccia utente che spesso non ha bisogno dell'intero record. Solo la generazione del report avviene tramite una richiesta HTTP, in quanto è necessario restituire soltanto il documento generato.
2. Il software gestionale del laboratorio di analisi invece effettua richieste HTTP presso degli *endpoint* appositi; la scelta di usare il protocollo HTTP è dovuta sia al supporto più diffuso, trattandosi di un servizio esterno, che al fatto che questo client necessita sempre dello stesso tipo di rappresentazione delle risorse.

4.2.3 Model, View, Controller

Il pattern architetturale adottato per la progettazione del *back end*, in accordo all'architettura preesistente e alle convenzioni di Rails, è di tipo MVC. Esso è tradizionalmente utilizzato per lo sviluppo di applicativi desktop con interfaccia grafica, ma risulta molto popolare anche per lo sviluppo di applicazioni web. Il pattern MVC prevede di separare della logica dell'applicativo software in tre componenti interconnessi:

- **Model**: componente centrale del pattern che contiene la *business logic*, ovvero le regole di accesso e manipolazione dei dati su cui lavora l'applicativo; è costituito, generalmente, dalle rappresentazioni interne dei dati e dai metodi che riguardano tali rappresentazioni.
- **View**: componente corrispondente all'interfaccia dell'applicativo, si occupa di rendere visibili all'utente i dati e le operazioni permesse; contiene i vari elementi grafici dell'interfaccia e le rappresentazioni esterne dei dati.
- **Controller**: componente che contiene la *application logic*: accetta gli input immessi dall'utente tramite la **view**, gli elabora in operazioni da richiedere al **model** e restituisce un output che viene visualizzato sempre dalla **view**.

4.2.4 Implementazione del pattern MVC

Il pattern architetturale MVC viene implementato nella progettazione dell'applicativo nel modo seguente (N.B.: da qui, **{model/view/controller}** si riferisce rispettivo al componente del pattern MVC, mentre *{model/view/controller}* si riferisce alla rispettiva classe di oggetti).

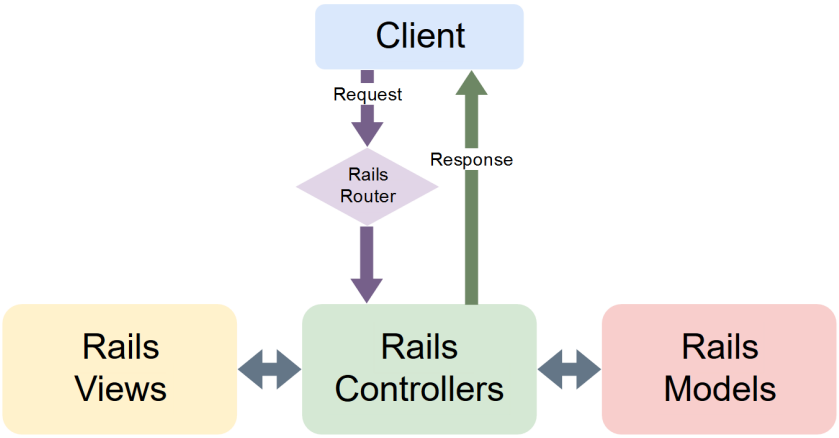

Figura 4.3: Implementazione del pattern MVC in Rails

4.2.4.1 Model

Il **model** è costituito dall'insieme dei *models* Active Record. Active Record è un modulo di Rails dedicato all'implementazione del **model** di un'applicazione sfruttando la tecnica detta *Object Relational Mapping*, che permette di connettere classi di oggetti di un'applicazione a tabelle in un database relazionale. Usando tale tecnica le proprietà e relazioni degli oggetti dell'applicazione possono essere conservate nel e lette dal database senza dover impiegare comandi SQL.

Ogni *model* è associato ad una tabella del database convenzionalmente attraverso il nome, che è la versione singolare del nome della tabella (e.g. il *model* `User` è automaticamente associato alla tabella `users`); ogni istanza di un *model* rappresenta una riga della tabella. Le relazioni tra tabelle sono esplicitate mediante metodi speciali, detti *associations*, nei modelli interessati. Active Record permette mette anche a disposizione metodi per effettuare *query* sui *model* così come si farebbe in su tabelle di un database.

Active Record	**Database**
Model	Tabella
Attributo	Colonna
Istanza di un *model*	Riga
Association	Relazione

Tabella 4.2: Corrispondenza tra Active Record e database.

La manipolazione delle istanze dei *model* prevede due fasi: l'assegnazione di eventuali nuovi valori agli attributi di un'istanza, che avviene normalmente tramite assegnazioni, e il passaggio delle modifiche a database mediante uno dei seguenti metodi di Active Record:

- **`save`**: salva le modifiche apportate ai valori dell'istanza a database, attivando eventuali vincoli;
- **`update`**: ha la stessa funzione di `save`,ma in questo caso i nuovi valori possono essere passati come parametri al metodo stesso;
- **`destroy`**: le modifiche vengono ignorate e il record viene rimosso dal database.

A questi metodi può venire associata l'esecuzione automatica di *callbacks*, che agiscono da vincoli o apportando ulteriori modifiche.

Gli attributi di un *model* sono automaticamente fatti corrispondere alle colonne della tabella, sempre attraverso la nomenclatura. Sui valori di tali attributi sono attivi i vincoli imposti a database, più ulteriori vincoli definiti nel *model* nella forma di metodi di validazione: al salvataggio di una nuova istanza di un *model*, verrà prima validato secondo i metodi specificati, e poi validato a database; un fallimento in uno dei due processi di verifica ne previene la scrittura a database.

I *model* accentrano anche tutta la *business logic* dell'applicativo: tutti i metodi relativi alla manipolazione dei dati di un certo tipo di oggetto vanno inseriti nel corrispettivo *model*.

4.2.4.2 Controller

Il **controller** è costituito dall'insieme dei *controllers* definiti con Active Controller. Active Controller è un modulo di Rails che permette di applicare il principio CoC alla creazione di controller per l'applicazione. In generale un *controller* riceve una richiesta HTTP, esegue il proprio metodo corrispondente all'interazione desiderata con il **model** e usa una *view* per creare un output. La corrispondenza tra ciascun *endpoint*, *controller* e metodo del *controller* viene semplicemente specificata all'interno di uno speciale file di *routing*; il resto della configurazione è trasparente allo sviluppatore. La progettazione dell'applicativo prevede un *controller* per ogni attore esterno.

4.2.4.3 View

La **view** è costituita dall'insieme delle *views* definite tramite la gemma Jbuilder. Jbuilder permette di definire una struttura dati JSON generale all'interno, che può quindi essere "riem-

pita" con i dati necessari di volta in volta. Tali *view* vengono utilizzate come *body* delle risposte HTTP date dal **controller** in seguito all'elaborazione di una richiesta, e contengono le informazioni richieste in un formato concordato con il destinatario.

4.3 Struttura dell'applicazione

L'applicazione è stata strutturata in modo da riflettere la separazione concettuale dei componenti prevista dalle scelte architetturali e rispettando le linee guida fornite da Rails. Le cartelle di maggior interesse sono le seguenti:

- Nel percorso ***root*** si trovano i file di configurazione di progetto richiesti da Rails, tra cui il `Gemfile` che specifica la versione di Ruby e le gemme richieste.
- Nella cartella **`app`** risiede il cuore dell'applicativo, ovvero tutti i file integrali al suo funzionamento per quanto riguarda la logica di programmazione: i vari *models*, *views* e *controllers*, le definizioni del sistema dei tipi di GraphQL, i file che descrivono i componenti del pannello di amministrazione, il *mailer* e i servizi aggiuntivi (logger, client per SNS e spedizioni, gestore dei codici OTP).
- La cartella **`config`** contiene i file di configurazione dell'applicativo: di rilevanza sono il file di *routing*, i file di configurazione dell'esecuzione del server, il file di definizione degli *environment* di sviluppo e i file contenenti le credenziali. Questi ultimi sono crittografati automaticamente da Rails, permettendone l'accesso solo al codice in esecuzione sul server tramite un apposito metodo.
- La cartella **`db`** contiene le varie *migrations* con la storia delle modifiche allo schema del database e i *seeds* di inizializzazione.
- Infine, la cartella **`spec`** contiene i vari test di unità e di integrazione, nonché varie classi *helper* di supporto all'esecuzione dei test.

4.3.1 Database

Il database dell'applicazione è un database relazionale implementato con PostgreSQL, un DBMS *open source* basato sul linguaggio SQL. PostgreSQL non viene mai usato direttamente dallo sviluppatore se non nella fase iniziale di configurazione; Rails permette infatti di gestire il database attraverso un DSL specializzato, che non richiede quindi di scrivere alcuna riga di codice SQL. PostgreSQL è stato configurato come DBMS di default per l'applicazione Rails e il database "vero e proprio" viene creato, gestito e acceduto in modo trasparente

allo sviluppatore, eccezion fatta per situazioni di emergenza in cui sia necessario intervenire manualmente (e.g. per ripopolare il database dopo averlo svuotato).

4.3.1.1 Migrations

Le *migrations* sono dei file realizzati con un DSL specifico, facente parte del modulo Active Record di Rails, che contengono alterazioni allo schema del database dell'applicazione. In una *migration* vengono descritte le operazioni di manipolazione dello schema che si desidera effettuare. Le *migrations* vengono create attraverso una procedura di generazione automatizzata (*generator*) fornita da Active Record; il nome di ognuna è costituito dal *timestamp* in formato `YYYYMMDDHHMMSS` seguito da un nome descrittivo dell'operazione eseguita. Questo permette di effettuare un versionamento delle modifiche allo schema del database: con il comando `rails db:migrate` vengono eseguite tutte le *migrations* in coda di esecuzione, mentre con `rails db:rollback` è possibile annullare le ultime modifiche.

4.4 Models

In questa sezione verranno descritti i *models* che risultano essere di principale interesse per la progettazione dell'applicativo. Tutti i *models* sono situati nella stessa cartella e non viene applicato alcun tipo di gerarchia ad essi. È stato implementato un *model* per ogni entità da gestire (rappresentata a database con una tabella), più alcuni *models* speciali associati a tabelle di *join*, il cui scopo è puramente di riflettere le relazioni molti a molti di altre tabelle. Ogni *model* presenta, di default, 3 attributi:

- **`id`**: codice identificativo interno di un'istanza e chiave primaria a database;
- **`created_at`**: data e orario di creazione dell'istanza;
- **`updated_at`**: data e ora dell'ultima modifica dell'istanza;

N.B.: verranno fornite solo le informazioni rilevanti all'applicazione "Biota", il codice condiviso non utilizzato sarà ignorato.

4.4.1 Session

4.4.1.1 Descrizione

Model centrale al funzionamento dell'applicazione, rappresenta una sessione di esame associata ad un utente ed è il *model* più corposo per numero di attributi, metodi e relazioni con altri *models*. Ogni istanza di `Session` corrisponde ad una diversa sessione di esame attiva per un cliente presso una certa farmacia, e contiene le informazioni relative al questionario, al campione, allo stato di avanzamento dell'esame e alla diagnosi.

I metodi di questo *model* riguardano principalmente l'aggiornamento del questionario, la gestione dello stato di avanzamento dell'esame e vincoli di integrità non inseribili a database.

L'integrazione con la piattaforma esistente richiede una discriminazione delle sessioni a seconda del servizio a cui appartengono; essa avviene grazie ad una relazione uno a molti con il *model* `Service`, che verrà tuttavia trascurato in quanto di interesse marginale per la progettazione.

4.4.1.2 Relazioni

Model	Relazione	Spiegazione
`Customer`(alias di `User`)	Uno a molti	Un esame appartiene ad un solo cliente, un cliente può richiedere più esami.
`Pharmacy`	Uno a molti	Un esame è effettuato presso una sola farmacia, la quale può avviare più esami.
`QuestionnaireTemplate`	Uno a molti, opzionale	Un esame può usare un solo *template* di questionario, il quale viene condiviso da più esami.
`QuestionnaireAnswer`	Molti a uno	Il questionario di un esame può comprendere più risposte, ognuna delle quali appartiene solo a quell'esame .

`Shipment`	Uno a molti, opzionale	Il campione di un esame può appartenere ad una sola spedizione, la quale comprende più campioni.

Tabella 4.3: Tabella delle relazioni del *model* `Session`.

4.4.1.3 Attributi

Nell'elencare gli attributi, vengono omesse la chiave primaria `id` (comune a tutti i *model*) e le chiavi esterne di altri *model*, deducibili dalla tabella delle relazioni.

Attributo	Tipo	Descrizione
`status`	`enum`	Stato di avanzamento dell'esame, impostato di default a "started".
`questionnaire_updated_at`	`datetime`	Data e ora di ultima compilazione del questionario.
`sample_code`	`string`	Codice univoco di identificazione del campione associato all'esame; deve appartenere ad una lista fornita dal cliente.
`sample_status`	`enum`	Stato del campione associato ad un esame, impostato di default a "requested".
`sample_ready_at`	`datetime`	Data e ora di ricezione del campione dal paziente.
`sample_dispatched_at`	`datetime`	Data e ora di spedizione del campione associato all'esame.

report	`Attachment`	Referto delle analisi in formato PDF.
report_available_at	`datetime`	Data e ora di ricezione del referto delle analisi.
notes	`text`	Note allegate al referto delle analisi dal laboratorio.
gastroenterologist_notes_general	`text`	Diagnosi generale del gastroenterologo.
gastroenterologist_notes_specific	`text`	Diagnosi specifica del gastroenterologo.
active_substance	`string`	Principio attivo consigliato.
diet_table	`[text]`	Tabella della dieta (una rappresentazione *color-coded* del beneficio apportato da certi alimenti).

Tabella 4.4: Tabella degli attributi del *model* `Session`.

4.4.1.4 Metodi

Metodo	Descrizione
add_questionnaire_answer	Aggiunge una nuova risposta alla compilazione del questionario.
add_questionnaire_open_answer	Aggiunge una nuova risposta aperta alla compilazione del questionario.
check_if_sample_ready	*Callback* che verifica se il campione è stato consegnato e aggiorna l'attributo `sample_ready_at`.

check_if_sample_dispatched	*Callback* che verifica se il campione è stato spedito e aggiorna l'attributo `sample_dispatched_at`.
default_diet_table	Imposta la tabella della dieta di default.
process_report	Invoca il servizio di generazione del referto dettagliato, con l'opzione di omettere le informazioni personali del paziente.
status_consistency	*Callback* di validazione, verifica che lo stato di avanzamento dell'esame sia consistente con lo satto del campione.
sample_code_consistency	*Callback* di validazione, verifica che il codice campione sia nella lista di quelli approvati.
selected_questionnaire_consistency	*Callback* di validazione, verifica che il *template* del questionario sia tra quelli disponibili.
sample_in_lab?	Verifica se il campione è stato ricevuto dal laboratorio.

Tabella 4.5: Tabella dei metodi del *model* `Session`.

4.4.2 User

4.4.2.1 Descrizione

Altro *model* centrale all'applicativo, rappresenta un utente; l'integrazione con il codice esistente ha richiesto di renderlo un *model* polivalente, in grado di rappresentare sia le tre categorie di utenti autorizzati che i clienti. Per questo si trova ad avere un gran numero di attributi, che vengono verificati in modo diverso a seconda del valore dell'attributo `role` (che agisce da discriminante tra le tipologie di `User`). Gli utenti previsti sono "customer" (clienti), "pharmacy" (farmacisti), "gastroenterologist" (gastroenterologi affiliati) e "bmr" (supervisori del laboratorio).

I metodi di questo *model* riguardano la gestione della password (per utenti autorizzati), la generazione e verifica di codici OTP (per i clienti) e la visibilità delle sessioni a seconda del ruolo.

4.4.2.2 Relazioni

La relazione molti a molti che gli `User` ruolo "customer" hanno con il *model* `Pharmacy` viene rappresentata a database mediante una *join table*, che viene riflessa nella progettazione dal rispettivo *model*, `CustomerMembership`.

Model	Relazione	Spiegazione
`Pharmacy`	Uno a molti, opzionale	Valida solo per farmacisti; un farmacista è associato ad una sola farmacia, che ne può impiegare più di uno.
`CustomerMembership`	Molti a uno	Valida solo per clienti; un cliente può avere più "iscrizioni" a farmacie diverse, mentre ogni iscrizione è esclusiva ad un cliente.
`Pharmacy`, indiretta	Molti a molti	Valida solo per clienti; un cliente può essersi registrato presso più farmacie, ogni farmacia può avere più clienti.

Tabella 4.6: Tabella delle relazioni del *model* `User`.

4.4.2.3 Attributi

Gli attributi, in questo particolare caso, sono quasi totalmente diversi tra clienti e utenti autorizzati; purtroppo non è stata possibile effettuare una separazione in due *model* diversi per non compromettere il funzionamento del *back end* preesistente, quindi tali attributi sono tutti raggruppati in `User` e verificati con appositi *callbacks*. Gli attributi che descrivono le generalità del cliente sono raggruppati per brevità. Gli attributi esclusivi ad una categoria di utenti saranno indicati dalle lettere [C], [P], [G] e/o [B] (rispettivamente: "customer", "pharmacy", "gastroenterologist", "bmr") all'inizio della descrizione.

Attributo	Tipo	Descrizione
`role`	`enum`	[C,P,G,B] Ruolo assegnato all'utente.
`email`	`string`	[C,P,G,B] Indirizzo email dell'utente.
Generalità	*Vari*	[C] Insieme degli attributi che descrivono le informazioni personali del cliente, inserite in fase di registrazione. Ne fa parte l'attributo **`mobile_number`**.
`encrypted_password`	`string`	[P,G,B] Password dell'utente crittografata da Devise.
`reset_password_token`	`string`	[P,G,B] Token per il reset della password fornito da Devise.
`reset_password_sent_at`	`datetime`	[P,G,B] Data e ora di invio del link di reset della password.
Attributi di **sign in**	Vari	[P,G,B] Insieme di attributi che tracciano gli accessi effettuati.
`tokens`	`json`	[P,G,B] Insieme di token che descrivono una sessione dell'utente, generati da Devise. Vengono usati per autenticare le richieste GraphQL.
`privacy_consent`	`boolean`	[C] Consenso al trattamento dei dati.
`privacy_policy`	`Attachment`	[C] Copia del modulo per il consenso al trattamento dei dati firmato.
`privacy_accepted_at`	`datetime`	[C] Data e ora della conferma del consenso al trattamento dei dati OTP.
`otp_secret_key`	`string`	[C] Chiave segreta per la generazione di codici OTP per l'utente.
`otp_counter`	`integer`	[C] Contatore dei tentativi di invio del codice OTP.
`last_otp_at`	`datetime`	[C] Data e ora dell'ultimo invio del codice OTP.

Tabella 4.7: Tabella degli attributi del *model* `User`.

4.4.2.4 Metodi

Metodo	Descrizione
`sessions`	[C,P,G,B] Restituisce le istanze di `Session` associate all'utente, restringendone lo *scope* a seconda della categoria di utente.
`password_complexity`	[P,G,B] Verifica che venga rispettata la complessità minima per la password.
`generate_random_password`	[P,G,B]Genera una password casuale per l'utente.
`generate_privacy_otp`	[C] Genera un nuovo codice OTP l'accettazione della *privacy policy*, crea un istanza di `SMS::Client` e ne richiama il metodo per inviare un nuovo messaggio contenente il codice al cliente.
`verify_privacy_consent`	[C] Se viene fornito un codice OTP come argomento, ne verifica la validità ed aggiorna il consenso alla *privacy policy*; altrimenti, verifica se `privacy_policy` contiene un file allegato ed aggiorna il consenso.
`sms_limit_cooldown`	[C] Verifica che non sia stato superato il limite di tentativi giornalieri di invio SMS, ed eventualmente impedisce l'invio.
`has_mobile_number`	[C] *Callback* di validazione, verifica che se l'`User` è un cliente allora abbia un numero di telefono valido.

Tabella 4.8: Tabella dei metodi del *model* `User`.

4.4.3 AdminUser

4.4.3.1 Descrizione

Rappresenta un utente con privilegi di amministrazione. Si tratta di un *model* separato da `User`, e contiene soltanto gli attributi necessari all'autenticazione con Devise: `email`, `encrypted_password` e i vari attributi legati al reset della password e al conteggio degli accessi.

4.4.4 Pharmacy

4.4.4.1 Descrizione

Rappresenta una farmacia affiliata.

Si tratta di un *model* piuttosto semplice, che contiene semplicemente i dati della farmacia e le informazioni utili alla spedizione; è però fondamentale per discriminare quali `Session` debbano essere visibili ad un farmacista, dato che esami svolti presso farmacie diverse devono essere accessibili solo dai dipendenti delle rispettive farmacie. Questo *model* non contiene alcun metodo di nota, soltanto un controllo sul formato del valore dell'attributo `province`.

4.4.4.2 Relazioni

Model	Relazione	Spiegazione
`Pharmacist` (alias di `User`)	Molti a uno	Una farmacia può impiegare più farmacisti, un farmacista può essere impiegato presso una sola farmacia.
`CustomerMembership`	Molti a uno	Una farmacia può avere più "iscrizioni" da clienti diversi, mentre ogni iscrizione è esclusiva ad una farmacia.
`Customer`, indiretta	Molti a molti	Una farmacia può avere più clienti, un cliente può registrarsi presso più farmacie.
`Session`	Molti a uno	Una farmacia può avviare più esami, un esame viene fatto presso una sola farmacia.

`Shipment`	Molti a uno	Una farmacia può fare più spedizioni, una spedizione proviene da una sola farmacia.

Tabella 4.9: Tabella delle relazioni del *model* `Pharmacy`.

4.4.4.3 Attributi

Gli attributi di questo *model* sono di poco interesse ai fini della progettazione in quanto sono principalmente descrittivi. Possono essere raggruppati in due categorie:

- **Dettagli**: Insieme di attributi che specificano dettagli della farmacia come logo (allegato), nome, orario di apertura, contatti, etc.;
- **Indirizzo per la spedizione**: Insieme di attributi che descrivono gli estremi per effettuare spedizioni, specificando l'indirizzo della sede della farmacia.

4.4.5 CustomerMembership

4.4.5.1 Descrizione

Rappresenta la *join table* tra i *models* `User` (con alias `Customer` per indicare che è ristretta ai clienti) e `Pharmacy` nella forma di "iscrizione" di un cliente ad una farmacia.

È un model "vuoto", contenente solo gli attributi di default e le chiavi esterne ai due *model* coinvolti (ovvero due attributi che si riferiscono ai rispettivi campi `id`).

Model	Relazione	Spiegazione
Customer (alias di `User`)	Uno a molti	Un'iscrizione proviene da un solo cliente, il quale può iscriversi più volte.
`Pharmacy`	Uno a molti	Un'iscrizione è presso una sola farmacia, che può ricevere più iscrizioni.

4.4.6 QuestionnaireTemplate

4.4.6.1 Descrizione

Rappresenta un *template* di questionario, ovvero un insieme di domande e possibili risposte.

Serve semplicemente a raggruppare le domande in *template* che possono essere associati a degli esami; anche questo dunque è un *model* principalmente vuoto.

4.4.6.2 Relazioni

Model	Relazione	Spiegazione
`Question`	Molti a uno	Un questionario può prevedere più domande, ogni domanda appartiene ad un solo *template* di questionario.

Tabella 4.11: Tabella delle relazioni del *model* `QuestionnaireTemplate`.

4.4.6.3 Attributi

Attributo	Tipo	Descrizione
`name`	`string`	Nome del *template*.
`code`	`string`	Codice univoco assegnato al *template*.

Tabella 4.12: Tabella degli attributi del *model* `QuestionnaireTemplate`.

4.4.7 Question

4.4.7.1 Descrizione

Rappresenta una domanda che può comparire in un questionario.

Le `Question` sono definite una sola volta e sono quindi riferite dalle effettive risposte date al questionario, dato che il testo della domanda non varia. Sono previsti tre tipi di domanda: “open” (a risposta aperta), “single_choice” (a risposta singola) e “multi_choice” (a risposta multipla). Il tipo di domanda determina i vincoli attivi sul valore della risposta e la modalità di elaborazione del testo della risposta.

4.4.7.2 Relazioni

Model	Relazione	Spiegazione
`QuestionnaireTemplate`	Uno a molti	Una domanda può appartenere ad un solo *template*, che prevede una o più domande.
`PossibleAnswers`	Molti a uno	Una domanda può avere più risposte previste, una risposta può appartenere ad una sola domanda; le domande aperte non hanno nessuna `PossibleAnswer`.

Tabella 4.13: Tabella delle relazioni del *model* `Question`.

4.4.7.3 Attributi

Attributo	Tipo	Descrizione
`kind`	`enum`	Tipo di domanda (a risposta aperta, singola o multipla).
`mandatory`	`boolean`	Obbligatorietà della domanda, di default a `True`.
`position`	`integer`	Posizione nell'ordine del questionario.
`text`	`string`	Testo della domanda.

Tabella 4.14: Tabella degli attributi del *model* `Question`.

4.4.7.4 Metodi

Metodo	Descrizione
`fix_positions`	*Callback* eseguito dopo eventuali modifiche al valore dell'attributo `position`, riordina le domande nel *template* associato.
`question_expected_answers`	*Callback* che verifica che le `PossibleAnswers` associate rispettino il tipo di domanda (le domande aperte non hanno alcuna `PossibleAnswer`).

Tabella 4.15: Tabella dei metodi del *model* `Question`.

4.4.8 PossibleAnswer

4.4.8.1 Descrizione

Rappresenta una possibile risposta ad una domanda, ovvero il testo, valore e posizione di una delle risposte selezionabili.

Un insieme di istanze di `PossibleAnswer`, associate alle relative `Question` appartenenti ad uno stesso `QuestionnaireTemplate`, va a costituire un questionario che viene visualizzato dall'utente.

4.4.8.2 Relazioni

Model	Relazione	Spiegazione
`Question`	Uno a molti	Una possibile risposta può appartenere ad una sola domanda, la quale può avere più risposte previste.

Tabella 4.16: Tabella delle relazioni del *model* `PossibleAnswer`.

4.4.8.3 Attributi

Attributo	Tipo	Descrizione
`position`	`integer`	Posizione nell'ordine delle risposte alla domanda associata.
`text`	`string`	Testo della risposta.
`value`	`decimal(8,3)`	Eventuale valore numerico della risposta.

Tabella 4.17: Tabella degli attributi del *model* `PossibleAnswer`.

4.4.8.4 Metodi

Metodo	Descrizione
`fix_positions`	*Callback* eseguito dopo eventuali modifiche al valore dell'attributo `position`, riordina le risposte possibili nella domanda associata.

Tabella 4.18: Tabella dei metodi del *model* `PossibleAnswer`.

4.4.9 QuestionnaireAnswer

4.4.9.1 Descrizione

Rappresenta una risposta data al questionario di un esame, ovvero il testo, valore e posizione della risposta selezionata dall'utente.

La relazione tra questo *model* e i *model* che descrivono domande è differente rispetto a quella di `PossibleAnswers`: le istanze dei tre *model* descritti in precedenza descrivono la struttura di un questionario; un insieme di istanze di `QuestionnaireAnswer`, invece, va a costituire la effettiva compilazione di un questionario.

Una istanza `QuestionnaireAnswer` è composta, rispettivamente, da puro testo se la domanda associata è a risposta aperta (in questo caso l'attributo `denormalized_text` viene assegnato direttamente), oppure dall'insieme di testi e valori delle `PossibleAnswer` scelte nella compilazione se la domanda è a risposta singola o multipla.

4.4.9.2 Relazioni

Model	Relazione	Spiegazione
`Session`	Uno a molti	Una risposta può essere data solo alla compilazione del questionario per un esame, il cui questionario associato prevede di dare più risposte.
`QuestionnaireTemplate`	Uno a molti	Una risposta è associata solo ad un *template* di questionario, per cui possono essere compilate più risposte.
`Question`	Uno a molti	Una risposta è data solo ad una domanda, cui possono essere associate più risposte (domande a risposta multipla, o risposte alla stessa domanda date in esami separati).

`PossibleAnswer`	Molti a molti	Una risposta può risultare dalla selezione di più risposte possibili, e una risposta possibile può essere selezionata in più compilazioni diverse.

Tabella 4.19: Tabella delle relazioni del *model* `QuestionnaireAnswer`.

4.4.9.3 Attributi

Attributo	Tipo	Descrizione
`denormalized_text`	`string`	Testo della risposta, denormalizzato.
`denormalized_value`	`string`	Valore della risposta, denormalizzato.
`denormalized_question_text`	`string`	Testo della domanda, denormalizzato.
`denormalized_question_position`	`string`	Posizione della domanda, denormalizzata.

Tabella 4.20: Tabella degli attributi del *model* `QuestionnaireAnswer`.

4.4.9.4 Metodi

Metodo	Descrizione
`denormalize`	*Callback* eseguito prima di ogni validazione per risposte a domande a scelta singola o multipla, denormalizza i testi ed i valori delle `PossibleAnswer` associate e li assegna ai rispettivi attributi `denormalized` (idem per gli attributi rilevanti della `Question` associata).
`get_answer`	Restituisce il valore o testo della risposta, a seconda di quale sia rilevante.

Tabella 4.21: Tabella dei metodi del *model* `QuestionnaireAnswer`.

4.4.10 Shipment

4.4.10.1 Descrizione

Rappresenta una spedizione organizzata da una farmacia. Una spedizione è composta da un'insieme di campioni che si desidera inviare al laboratorio, ed è caratterizzata dalla data in cui viene richiesta e dalla data di ritiro effettiva.

L'iter operativo per effettuare una spedizione è il seguente: la spedizione viene creata, vengono aggiunti i campioni, quindi viene prenotata tramite le API del corriere SDA e viene stampata la lettera di vettura.

I metodi di questo *model* dunque validano che le date siano consistenti con i vincoli di SDA ed inviano le richieste necessarie alle API del corriere tramite il client implementato.

4.4.10.2 Relazioni

Model	Relazione	Spiegazione
`Pharmacy`	Uno a molti	Una spedizione proviene da una sola farmacia, una farmacia può richiedere più spedizioni.
`Session`	Molti a uno	Una spedizione comprende i campioni di più esami, un campione appartiene ad una sola spedizione.

Tabella 4.22: Tabella delle relazioni del *model* `Shipment`.

4.4.10.3 Attributi

Attributo	Tipo	Descrizione
`booking`	`string`	Codice di prenotazione.
`request_date`	`date`	Data di richiesta della spedizione.
`shipment_date`	`date`	Data di ritiro della spedizione.
`sender_notes`	`string`	Eventuali note del mittente.
`status`	`enum`	Stato di avanzamento della spedizione; non pienamente utilizzato, è stato inserito prevedendo di aggiungere funzionalità di tracking della spedizione.

waybill	`Attachment`	Lettera di vettura.
waybill_code	`string`	Codice della lettera di vettura.

Tabella 4.23: Tabella degli attributi del *model* `Shipment`.

4.4.10.4 Metodi

Metodo	Descrizione
book_shipment	Prenota il ritiro della spedizione, creando un'istanza di `SDA::Client` e richiedendo l'invio di una richiesta di prenotazione; in caso di successo, aggiorna la spedizione con il numero di prenotazione ricevuto.
get_waybill	Richiede la lettera di vettura della spedizione, creando un'istanza di `SDA::Client` e richiedendo l'invio di una richiesta per ottenere il documento; in caso di successo, aggiorna la spedizione allegando il documento all'istanza.
get_answer	Restituisce il valore o testo della risposta, a seconda di quale sia rilevante.

Tabella 4.24: Tabella dei metodi del *model* `QuestionnaireAnswer`.

4.4.11 ApprovedCode

4.4.11.1 Descrizione

Rappresenta un codice campione approvato dal cliente. È un *model* vuoto, ovvero possiede solo l'attributo `sample_code` utilizzato per controllare che il codice campione associato ad un esame sia valido.

4.4.12 Account

4.4.12.1 Descrizione

Rappresenta un account di un servizio esterno a cui è permesso di inviare richieste HTTP, e contiene i dati necessari a verificare le credenziali inviate. Si tratta di un *model* creato per sup-

portare la funzionalità di autenticazione delle richieste HTTP, che purtroppo non è giunta in produzione.

4.4.12.2 Attributi

Attributo	Tipo	Descrizione
`name`	`string`	Nominativo dell'account.
`access_id`	`string`	Identificativo di accesso.
Generalità	*secret*	Chiave segreta di generazione delle credenziali.

Tabella 4.25: Tabella degli attributi del *model* `Account`.

4.4.12.3 Metodi

Metodo	Descrizione
`generate_secret`	*Callback* eseguito alla creazione di un account, genera una nuova chiave segreta di autenticazione.
`generate_access_id`	*Callback* eseguito alla creazione di un account, genera l'identificativo di accesso elaborando la funzione hash MD5 del nominativo più un numero casuale.

Tabella 4.26: Tabella dei metodi del *model* `Account`.

4.5 Views

Le *views* possiedono un ruolo marginale nell'applicativo, essendo principalmente costituite da file `.jbuilder` per la generazione dei *body* delle risposte HTTP date dai controller.

Le *views* risiedono nell'omonima cartella, e sono organizzate in sottocartelle che rispecchiano l'organizzazione dei controller: il percorso relativo della cartella contenente le *views* per un *controller* segue il percorso relativo di quest'ultimo; la cartella possiede lo stesso nome del controller (senza la dicitura `_controller` alla fine). Ad esempio, le *views* per il controller **`sample_session_controller`** con path relativo `controllers/`**`api`** sono situate nella cartella `views/`**`api/sample_session`**. Il nome del file corrispondente a ciascuna *view* è lo

stesso della funzione del *controller* che ne richiede il *rendering*.

Di seguito vengono semplicemente elencate le *views* realizzate e la loro funzione:

Nome	Descrizione
`_data`	*View* parziale che rappresenta i dati di un esame rilevanti per il laboratorio, quali codice campione, risposte al questionario e nome della farmacia.
`_sample`	*View* parziale che rappresenta i dati di un campione (codice, stato e ultimo aggiornamento).
`check_in`	*View* di conferma in risposta alla richiesta di check-in di un campione.
`error`	*View* di errore, descrive il motivo del rifiuto della richiesta.
`reject`	*View* di conferma in risposta alla richiesta di rifiuto di un campione.
`show`	*View* di risposta alla richiesta di visualizzazione dei dati di un esame.
`start`	*View* di conferma in risposta alla richiesta di inizio delle analisi di un campione.
`success`	*View* di conferma in risposta alla richiesta di invio del referto delle analisi di un campione.

Tabella 4.27: Elenco delle *views*.

4.6 Controllers

In questa sezione verranno descritti i *controllers* che gestiscono le richieste HTTP inviate all'applicazione. Contrariamente allo standard per le applicazioni Rails, essi non sono stati progettati per gestire le classiche operazioni CRUD sulle risorse dell'applicativo, quanto per rispondere a delle richieste specifiche effettuate dai due client previsti.

I *controllers* sono strutturati in una gerarchia delle cartelle dal valore principalmente organizzativo; quelli di interesse, sviluppati nel corso del progetto, risiedono nella sottocartella `api`.

I *controllers* descritti ereditano da una classe preesistente, chiamata **`ApiController`**, che definisce dei metodi di inizializzazione e riconoscimento del client.

4.6.1 API::SampleSessionController

Gestisce le richieste in arrivo dal software gestionale del laboratorio, elaborandole e fornendo risposte adeguate. Per questo controller è stata sviluppata ed implementata la funzionalità di autenticazione HMAC delle richieste, che però non è entrata in produzione al termine del progetto.

Di seguito vengono riportati i metodi disponibili:

Metodo	Descrizione
`api_authenticate`	*Callback* eseguito alla ricezione di una richiesta, che verifica siano presenti le credenziali di autenticazione corrette.
`process_report`	Gestisce la richiesta di invio del referto dettagliato con la diagnosi del gastroenterologo per un esame; il referto viene generato senza includere le generalità del paziente per proteggerne l'anonimato.
`check_in`	Gestisce la richiesta di *check-in* di un campione (ovvero prima ricezione in laboratorio), ne aggiorna lo stato a "delivered"; accetta come parametro il codice del campione associato.
`reject`	Gestisce la richiesta di rifiuto di un campione, ne aggiorna lo stato a "rejected" e inserisce nelle note dell'esame corrispondente la ragione del rifiuto; accetta come parametro il codice del campione associato.
`show`	Gestisce la richiesta di invio dei dati di un esame; accetta come parametro il codice del campione associato.
`start`	Gestisce la richiesta di inizio delle analisi su un campione, ne aggiorna lo stato a "under_examination"; accetta come parametro il codice del campione associato.
`sync`	Gestisce la richiesta di sincronizzazione della lista di esami per cui è stato ricevuto un campione; accetta come parametro un *timestamp* che il limite temporale per la sincronizzazione dei cambiamenti.

Tabella 4.28: Metodi del *controller* `SampleSessionController`.

4.6.2 API::SessionController

Gestisce le richieste in arrivo dal *front end*, elaborandole e fornendo risposte adeguate. Devise autentica le richieste in automatico con i dati della sessione attiva inviati dal client; in aggiunta, viene verificato il ruolo dell'utente autenticato per determinarne i permessi.

Di seguito vengono riportati i metodi disponibili:

Metodo	Descrizione
`process_report`	Gestisce la richiesta di invio del referto dettagliato con la diagnosi del gastroenterologo per un esame, includendo le generalità a seconda del ruolo dell'utente. Accetta come parametro l'identificativo della `Session` corrispondente all'esame.
`process_report_anonymized`	Non gestisce direttamente una richiesta, viene invocato da `process_report` nel caso di una richiesta da un utente con ruolo diverso da *pharmacy*. Genera un referto dettagliato con la diagnosi ma senza le generalità del paziente.

Tabella 4.29: Metodi del *controller* `SessionController`.

4.6.3 GraphqlController

Esiste, inoltre, un ultimo *controller* di interesse: quello che si occupa di gestire le richieste che passano attraverso l'*endpoint* dedicato a GraphQL. L'autenticazione avviene, come per `API::SessionController`, da parte di Devise in modo automatizzato. Questo controller non è stato sviluppato come parte del progetto, ma viene riportato per completezza.

Metodo	Descrizione
`execute`	Metodo principale che gestisce le richieste effettuate attraverso l'*endpoint* dedicato; estrae i parametri e la *query* GraphQL dalla richiesta, li valida ed richiede l'esecuzione della *query* allo schema GraphQL definito, restituendone il risultato.

Tabella 4.30: Metodi del *controller* `GraphqlController`.

4.6.4 Routes

Di seguito vengono riportare le *routes* previste per effettuare richieste HTTP. I tratti salienti di ogni *route* sono tre: il metodo HTTP della richiesta, l'*endpoint* presso cui viene accettata e il metodo del *controller* che la elabora.

Nella descrizione degli *endpoint*, i parametri passati come parte dell'indirizzo sono indicati da un simbolo ":" prefisso.

HTTP	Endpoint	Metodo associato
GET	`/samples/:sample_code`	`show`
GET	`/samples/sync?:timestamp`	`sync`
GET	`/samples/:sample_code/report`	`process_report`
PUT	`/samples/:sample_code/check_in`	`check_in`
PUT	`/samples/:sample_code/reject`	`reject`
PUT	`/samples/:sample_code/start`	`start`
POST	`/samples/:sample_code/success`	`success`

Tabella 4.31: Elenco delle *routes* previste per il *controller* `SampleSessionController`.

HTTP	Endpoint	Metodo associato
GET	`/session/:session_id/report`	`process_report`

Tabella 4.32: Elenco delle *routes* previste per il *controller* `SessionController`.

HTTP	Endpoint	Metodo associato
GET	`/session/:session_id/report`	`process_report`

Tabella 4.33: Elenco delle *routes* previste per il *controller* `GraphqlController`.

4.7 API GraphQL

In questa sezione verrà descritta la progettazione della API GraphQL per la comunicazione con il *front end*. Grazie all'integrazione tra GraphQL e rails, per realizzare tali API è stato sufficiente definire il controller e il sistema dei tipi; il *runtime system* viene caricato all'avvio dell'applicativo.

L'utilizzo di GraphQL prevede prima di tutto di definire uno schema, ovvero un file che descrive la strutturazione del sistema dei tipi e come comportarsi nel caso venga riscontrato un errore. Non è quindi uno "schema" simile alla classica concezione applicabile ai database, quanto una prima definizione della logica della API.

Il sistema dei tipi si compone di tre parti: i *types* definiscono le rappresentazioni delle risorse, ovvero fanno corrispondere i vari *model* a delle rappresentazioni compatibili con GraphQL; le *queries* definiscono le operazioni di lettura del database; infine le *mutations* descrivono le operazioni di modifica e interazione con il database, come la creazione di una risorsa o l'operazione di reset della password.

Vengono di seguito elencati i tipi definiti, corredati di una breve descrizione.

4.7.1 Types

Nome	Descrizione
`ColourEnumType`	Tipo enumerativo per i colori della tabella della dieta.
`ImageType`	Tipo per la rappresentazione di immagini.
`MutationType`	Descrive tutte le *mutations* visibili, associandole alle relative definizioni.
`PharmacyType`	Tipo per la rappresentazione di un *model* `Pharmacy`.
`PossibleAnswerType`	Tipo per la rappresentazione di un *model* `PossibleAnswer`.
`QueryType`	Descrive tutte le *queries* visibili, associandole alle relative definizioni.

`QuestionType`	Tipo per la rappresentazione di un *model* `Question`.
`QuestionKindEnumType`	Tipo enumerativo per la rappresentazione della tipologia di una domanda.
`QuestionnaireAnswerType`	Tipo per la rappresentazione di un *model* `QuestionnaireAnswer`.
`QuestionnaireTemplateType`	Tipo per la rappresentazione di un *model* `QuestionnaireTemplate`.
`RoleEnumType`	Tipo enumerativo per la rappresentazione di un ruolo.
`SessionBaseType`	Tipo che implementa `SessionInterface`.
`SessionInterfaceType`	Interfaccia che definisce i campi per la rappresentazione di un *model* `Session` senza informazioni personali del cliente.
`SessionType`	Tipo che implementa `SessionInterface`, aggiungendo anche le informazioni relative al cliente.
`ShipmentType`	Tipo per la rappresentazione di un *model* `Shipment`.
`UserType`	Tipo per la rappresentazione di un *model* `User`.

Tabella 4.34: Elenco dei *types* GraphQL.

4.7.2 Queries

Nome	Descrizione
`AllCustomers`	Restituisce tutti gli utenti con ruolo "customer" iscritti alla farmacia dell'utente che effettua la richiesta.
`AllGastroenterologistSessions`	Restituisce tutti gli esami visibili a utenti con ruolo "gastroenterologist", senza le generalità dei pazienti.

`AllLabSessions`	Restituisce tutti gli esami visibili a utenti con ruolo “bmr”, senza le generalità dei pazienti.
`AllPharmacies`	Restituisce tutte le farmacie affiliate.
`AllSessions`	Restituisce tutti gli esami visibili a utenti con ruolo “pharmacist”, ovvero quelle effettuate presso la farmacia di impiego.
`AllShipments`	Restituisce tutte le spedizioni effettuate dalla farmacia dell’utente che effettua la richiesta.
`Me`	Restituisce l’utente autenticato.
`Session`	Restituisce i dettagli di un esame.
`SessionBySample`	Restituisce i dettagli di un esame, cercandolo con il codice campione.
`SessionWithSampleStatus`	Restituisce tutti gli esami con un certo stato di avanzamento.
`Shipment`	Restituisce i dettagli di una spedizione.
`User`	Restituisce i dettagli di un utente.
`UsersSamePharmacy`	Restituisce gli utenti autorizzati appartenenti alla stessa farmacia dell’utente corrente.

Tabella 4.35: Elenco delle *queries* GraphQL.

4.7.3 Mutations

Nome	Descrizione
`CustomerCreate`	Crea un nuovo cliente.
`CustomerDelete`	Rimuove un cliente.
`CustomerSendOtp`	Invia un codice OTP ad un cliente.
`CustomerUpdate`	Aggiorna un cliente.
`CustomerUploadPrivacyPolicy`	Carica una copia del consenso al trattamento dei dati per un cliente.

`CustomerVerifyOtp`	Verifica un codice OTP per un cliente.
`GastroenterologistInsertNotes`	Inserisce una diagnosi per un esame.
`PharmacyUpdate`	Aggiorna una farmacia.
`SessionCreate`	Crea un nuovo esame.
`SessionDelete`	Rimuove un esame.
`SessionSetDietTable`	Imposta la tabella della dieta per un esame.
`SessionUpdate`	Aggiorna un esame.
`ShipmentBooking`	Prenota una spedizione.
`ShipmentCreate`	Crea una nuova spedizione.
`ShipmentDelete`	Rimuove una spedizione.
`ShipmentUpdate`	Aggiorna una spedizione.
`ShipmentgetWaybill`	Richiede la lettera di vettura per una spedizione.
`UserChangePassword`	Cambia la password per un utente.
`UserCreate`	Crea un nuovo utente.
`UserDelete`	Rimuove un utente.
`UserResetPasswordRequest`	Richiede l'invio di una mail per il reset della password di un utente.
`UserResetPassword`	Effettua il reset della password di un utente.
`UserUpdate`	Aggiorna un utente.
`UserUploadLogo`	Permette all'utente di caricare un luogo per la propria farmacia di impiego.

Tabella 4.36: Elenco delle *mutations* GraphQL.

4.8 Services

Sebbene al cuore del funzionamento dell'applicativo vi siano i componenti MVC e le API descritte sopra, questi si appoggiano a dei servizi appositi per effettuare operazioni che si distaccano dalla "semplice" manipolazione del database.

Per la gestione di task complessi per cui sono richieste funzionalità che esulano dalle competenze di *models* e *controllers* sono quindi state realizzate delle classi ad-hoc che si comportano, appunto, come servizi che è possibile istanziare ed utilizzare. Questi risiedono nella cartella `services` e sono stati creati da zero; non si appoggiano quindi al *framework* Rails.

4.8.1 Generazione referto dettagliato

La generazione del referto dettagliato avviene *on-demand* a partire da quello offerto dal laboratorio, per varie ragioni: preservare al massimo la privacy del cliente, ridurre il carico di memoria occupata dal database ed adattarlo ad eventuali aggiornamenti con il minimo sforzo. Ciò tuttavia richiede di manipolare il file PDF originale, azione resa possibile dall'utilizzo della gemma HexaPDF.

È stato quindi progettata una classe apposita, `ReportProcessor`, che definisce i metodi di manipolazione del referto originale e lo modifica aggiungendo le generalità del paziente, la tabella della dieta consigliata, la diagnosi e la firma del gastroenterologo.

Per poter generare anche referti anonimizzati, queste aggiunte sono state separate in metodi modulari che si "rimbalzano" lo stesso file per ottenere solo le modifiche desiderate. L'unico attributo è una costante con il percorso dell'immagine con la firma del gastroenterologo.

Metodo	Descrizione
`initialize`	Inizializza un'istanza della classe con i dati dell'esame considerato, verificando che essi siano corretti e che il referto sia stato emesso.
`process`	Avvia la pipeline di generazione completa di tutte le modifiche e restituisce il file elaborato.
`process_anonymized`	Avvia la pipeline di generazione omettendo le generalità del cliente, e restituisce il file elaborato.
`report_with_patient`	Aggiunge le generalità del paziente.
`report_with_table`	Aggiunge la tabella della dieta.
`report_with_notes`	Aggiunge la diagnosi firmata del gastroenterologo.

Tabella 4.37: Metodi del *service* `ReportProcessor`.

In più sono presenti metodi privati di supporto al rendering della tabella della dieta, che

richiede di disegnare pallini di colore diverso per indicare quanto una categoria di alimenti sia adatta allo stile di vita del paziente.

4.8.2 Gestione dei codici OTP

Per gestire e verificare i codici OTP è stata utilizzata la gemma ROTP, che offre funzionalità di supporto sia per quanto riguarda la generazione che la verifica. Tuttavia, data la natura dei *models* e la necessità di associare univocamente codici e clienti, è stato progettata una classe apposita che aggiunge le funzionalità necessarie al *model* `User`.

4.8.2.1 OtpManager

Il modulo `OtpManager` viene incluso dal *model* `User` permettendo di definire l'attributo speciale `otp_secret_key`, che invece di essere un campo del database viene creato e gestito dall'istanza di `OtpManager`. Esso definisce la chiave segreta di generazione e verifica dei codici, associandone una univoca ad ogni istanza di cliente. Il modulo inoltre contiene dei metodi di inizializzazione che vengono eseguiti alla creazione di un'istanza del *model* e un metodo di verifica basato sull'istanza di *model* che lo esegue.

Metodo	Descrizione
`has_otp`	Aggiunge la colonna con l'attributo speciale al *model*, permettendo di definire lunghezza e validità temporale dei codici.
`otp_random_secret`	Genera la chiave segreta tramite la libreria ROTP.
`authenticate_otp`	Provvede alla verifica di validità di un codice, restituendo un valore booleano che ne riflette l;esito.
`otp_code`	Genera un nuovo codice per l'istanza di *model*.

Tabella 4.38: Metodi del *service* `OtpManager`.

4.8.3 Client per Amazon SNS

L'invio dei codici OTP per SMS avviene sfruttando un servizio cloud offerto da Amazon, AWS Simple Notification Service, noto anche come SNS. SNS permette di inviare messaggi di notifica personalizzati quasi istantaneamente ad un piccolo costo, ed è supportato dal SDK in Ruby offerto da Amazon per l'interazione con i propri servizi.

È stata progettata quindi la classe `SMS::Client`, che definisce il client di comunicazione con SNS. Si tratta di una classe molto snella, che offre solamente funzionalità di base, e implementa un pattern *adapter* per la classe `AWS::SNS::Client`. Il client è accoppiato ad un *logger* che tiene traccia dei messaggi inviati per scopi di debug.

Attributo	Descrizione
`client`	Istanza della classe `AWS::SNS::Client`.
`logger`	Istanza della classe `TimestampLogger`.

Tabella 4.39: Attributi del *service* `SMS::Client`.

Metodo	Descrizione
`send`	Invia un messaggio ad un numero adattando la relativa funzione di `AWS::SNS::Client` e richiamando il *logger* prima e dopo la richiesta di invio.

Tabella 4.40: Metodi del *service* `SMS::Client`.

4.8.4 Client per API di SDA

Si tratta del servizio più corposo, il corriere SDA si limita ad esporre degli *endpoint* di comunicazione ed è stato necessario implementare un client partendo da zero.

Dati i vari tipi di richieste previsti nella documentazione della API, e volendo lasciare l'implementazione aperta a future estensioni, è stato scelto di implementare una versione adattata del *command* pattern: è stato definito un modulo con una classe `SDA::Client` in grado di inviare delle richieste ed effettuare il *parsing* delle risposte ricevute. Il *command* pattern è stato però implementato solo in parte: infatti il client include necessariamente anche l'*invoker* dei comandi, che in questo caso è il modulo HTTP di Ruby; inoltre, dato che la procedura di invio è la stessa per ogni tipo di richiesta, anche l'esecuzione è delegata al client: attraverso un metodo privato, invocato dai metodi specifici per il tipo di richiesta, esso procede all'effettivo l'invio della richiesta. Questo perché le richieste differiscono per *endpoint* e contenuto, ma non per modalità di esecuzione.

Il funzionamento quindi è il seguente: il client vuole inviare un tipo di richiesta, la istanzia e ne delega l'esecuzione al metodo privato `send`; la risposta viene ritornata e ne viene

effettuato il *parsing*. Quindi il client restituisce i risultati emersi dalla risposta. Si tratta di una implementazione poco ortodossa, ma giudicata efficace per gli scopi che si desiderava raggiungere.

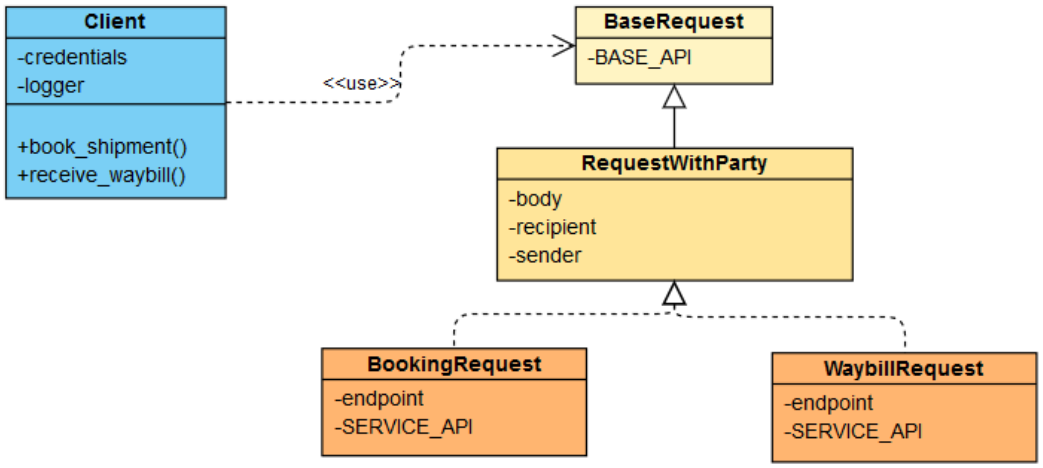

Figura 4.4: Diagramma UML delle classi

Grazie alla gerarchia di classi implementata per le richieste, eventuali modifiche agli *endpoint* sono coperte: ogni membro della gerarchia possiede definisce una parte dell'indirizzo con granularità man mano sempre più fine, partendo dall'URL base del server di SDA nella `BaseRequest` e arrivando all'URI specifico per la sottoclasse di richiesta desiderata; sono le implementazioni delle richieste finali a comporre l'*endpoint* in fase di inizializzazione, in modo che se viene cambiato ad un passo della gerarchia viene riflesso in tutte le sottoclassi in modo automatico.

4.8.4.1 SDA::Client

I metodi resi disponibili da questa classe eseguono le richieste invocando il metodo privato `send`, che le invia sfruttando il modulo HTTP e controlla la risposta cercando eventuali errori; in caso di esito negativo, ritorna la risposta al metodo che lo ha invocato continuando la normale esecuzione.

Attributo	Descrizione
`credentials`	Credenziali di accesso alle API SDA.
`logger`	Istanza della classe `TimestampLogger`.

Tabella 4.41: Attributi del *service* `SDA::Client`.

Metodo	Descrizione
`book_shipment`	Istanzia ed invia una richiesta `BookingRequest` di prenotazione di una spedizione, ne elabora la riposta e restituisce il codice di prenotazione.
`receive_waybill`	stanzia ed invia una richiesta `WaybillRequest` per la lettera di vettura di una spedizione, ne elabora la riposta e restituisce il codice della lettera di vettura e la lettera in formato PDF.

Tabella 4.42: Metodi del *service* `SDA::Client`.

4.8.4.2 SDA::Requests::BaseRequest

Classe di base, possiede solo l'attributo `BASE_API` che fornisce la prima parte dell'indirizzo a cui fare le richieste.

4.8.4.3 SDA::Requests::RequestWithParty

Classe che definisce la base per richieste in cui sono presenti mittente e ricevente; definisce una struttura dati e dei metodi privati per l'estrazione dei dettagli rilevanti dalla farmacia associata alla spedizione e dal file contenente le informazioni di spedizione del laboratorio. Questi metodi vengono invocati in fase di inizializzazione. Inoltre viene dichiarato un metodo virtuale `generate_body`, implementato dalle sottoclassi.

4.8.4.4 SDA::Requests::BookingRequest

Classe che definisce la richiesta di prenotazione di una spedizione; l'inizializzazione invoca il costruttore della superclasse, quindi l'implementazione del metodo `generate_body` ottenendo il *body* della richiesta a partire dalle strutture dati contenenti mittente e destinatario. Inoltre viene creato l'indirizzo completo dell'*endpoint* concatenando l'attributo ereditato `BASE_API` con l'attributo locale `SERVICE_API`.

4.8.4.5 SDA::Requests::WaybillRequest

Classe che definisce la richiesta di ottenimento della lettera di vettura per una spedizione; implementata in modo quasi identico a `BookingRequest`, varia soltanto per la diversa implementazione di `generate_body`, che viene adattata al tipo di richiesta.

4.8.5 Logger

Per facilitare le operazioni di debugging dei servizi sopracitati è stato implementato un *logger* in grado di essere istanziato specificando il file in cui salvare il *log* nonché di aggiungere un *timestamp* automaticamente all'evento loggato. Si tratta di un classe molto semplice, che eredita da `ActiveSupport::Logger` (classe *logger* resa disponibile dal modulo sActive Support) ridefinendone l'attributo `formatter` (che, appunto, formatta i messaggi registrati nel *log*) aggiungendo il *timestamp*. Il file di *log* viene specificato come parametro del costruttore.

4.9 Pannello di amministrazione

Il pannello di amministrazione è stato progettato ed implementato utilizzando ActiveAdmin, un framework compatibile con Rails che permette di realizzare interfacce amministrative minimizzando il codice necessario. La progettazione del pannello amministrativo, la cui versione base faceva parte del *back end* preesistente, ha richiesto semplicemente di aggiungere le schede richieste.

4.9.1 Struttura delle schede

Ogni scheda è associata ad un *model*, quindi ad un tipo di record del database; è inoltre possibile ridefinire e restringere lo *scope* dei record visibili da ciascuna scheda. Esse seguono tutte la stessa struttura:

- ***Model* di riferimento**: viene dichiarato quale *model* viene gestito, con la possibilità di specificarne un alias per chiarezza;
- **Priorità**: ordine di apparizione;
- **Parametri permessi**: vengono dichiarati i parametri modificabili; questo permette di ottenere un controllo a granularità molto fine sulle modifiche al database permesse;
- ***Scope***: viene dichiarato lo *scope* in forma di *query* sul *model* di riferimento;

- **Filtri**: vengono dichiarati eventuali filtri personalizzati;
- **Azioni**: vengono dichiarate eventuali azioni aggiuntive che sono disponibili per un record, ad esempio il download di un allegato;
- **Indice**: vengono dichiarati gli attributi mostrati nella tabella di indice per ogni record;
- ***Show***: vengono dichiarati gli attributi mostrati nella visualizzazione di un singolo record;
- ***Form***: viene dichiarato il form di creazione e modifica dei record, potenzialmente con alcuni campi esclusivi ad una delle due operazioni.

4.9.2 Elenco delle schede

Di seguito vengono elencate brevemente le schede realizzate o modificate nel corso del progetto:

Scheda	***Model***	**Azioni speciali**
Admin Users	`AdminUser`	Nessuna.
Clienti	`User` con role "customer"	Download dell'eventuale modulo di consenso alla *privacy policy* firmato.
Gastroenterologi	`User` con role "gastroenterologist"	Reset della password.
Farmacie	`Pharmacy`	Nessuna.
Questionari	`QuestionnaireTemplate`	Nessuna.
Sessioni	`Session`	Nessuna.
Utenti	`User` con role "pharmacist"	Nessuna.

Tabella 4.43: Tabella delle schede del pannello di amministrazione

Nulla facilisi. In vel sem. Morbi id urna in diam dignissim feugiat. Proin molestie tortor eu velit. Aliquam erat volutpat. Nullam ultrices, diam tempus vulputate egestas, eros pede varius leo.

Quoteauthor Lastname

5 Codifica e sviluppo

Nonostante la fase implementativa sia stata quella che ha richiesto più tempo nella realizzazione del progetto, occupando quasi 50% delle ore di lavoro, essa è semplicemente il risultato della fase di progettazione. In questo capitolo verranno presentati degli esempi di implementazione per i vari componenti progettati in modo da dare una miglior panoramica del lavoro svolto.

5.1 Norme di codifica

Il primo passo nella realizzazione di un prodotto di qualità consiste nell'adottare norme che minimizzino i rischi e gli errori che si possono verificare. Sono quindi state seguite delle precise norme di codifica volte a rendere il codice funzionale e facilmente comprensibile a sviluppatori diversi dall'autore.

Sono quindi state seguite pedissequamente le norme di codifica stabilite dalla community di Ruby on Rails, raccolte in una *style guide* pubblicamente disponibile. L'utilizzo di tali norme ha facilitato il processo di analisi statica che verrà esposto a breve.

5.2 Componenti MVC

5.2.1 Esempio di *model*

Osserviamo di seguito l'implementazione del *model* `Shipment`, uno degli ultimi ad essere implementato; esso presenta tutte le caratteristiche implementative interessanti comuni agli altri *models*.

```
require_relative '../services/sda/client'

class Shipment < ApplicationRecord
    belongs_to :pharmacy
    has_many :sessions, dependent: :nullify

    has_one_attached :waybill
    enum status: %i[pending booked in_transit delivered]

    validate :request_date_consistency
    validate :shipment_date_consistency
```

Listing 5.1: `shipment.rb`

Il codice inizia importando il client per le spedizioni e dichiarando la classe `Shipment` come sottoclasse di `ApplicationRecord`, permettendo quindi l'associazione automatica alla tabella `shipments` del database tramite ActiveRecord.

Subito dopo segue la definizione delle relazioni con gli altri *models*: `belongs_to` indica l'appartenenza univoca ad un altro *model* (quindi una relazione uno ad uno o uno a molti), mentre `has_many` specifica una relazione con cardinalità multipla. L'altro modello coinvolto completerà la definizione della relazione con la stessa sintassi.

Quindi osserviamo la definizione di attributi particolari: un `Attachment`, che non è definito come colonna della tabella del database ma la cui rappresentazione viene gestita da Rails, e un tipo enumerativo, rappresentato a database come un valore intero e fatto corrispondere alla relativa stringa dell'array.

Infine vengono definiti dei *callback* di validazione che eseguono dei controlli personalizzati quando viene creata o modificata un'istanza di `Shipment`.

```
    def receive_waybill
```

```
        client = SDA::Client.new
        waybill_file, waybill_code = client.receive_waybill(self)
        waybill.attach io: waybill_file, filename: "#{waybill_code}
            _ldv.pdf"
        self.waybill_code = waybill_code
        save!
    ensure
        waybill_file&.unlink
    end

    def book_shipment
        client = SDA::Client.new
        update! request_date: Date.today if request_date != Date.
            today
        booking = client.book_shipment(self)
        if booking.present?
            update! booking: booking, status: 'booked'
            sessions.each(&:set_as_dispatched)
        end
    end
```

Listing 5.2: `shipment.rb`

Una volta definiti relazioni, attributi e vincoli si passa alla codifica dei metodi pubblici esposti dalla classe: nell'esempio sono riportati i metodi `receive_waybill` e book_shipment, utilizzati rispettivamente per ottenere la lettera di vettura e prenotare una spedizione.

È di nota l'utilizzo di `save` e `update` per modificare il record associato all'istanza (la quale costituisce l'oggetto *this* implicito per l'invocazione dei suddetti metodi): nel primo caso, il metodo di ActiveRecord viene invocato per salvare le modifiche avvenute in precedenza; nel secondo vengono passati nuovi valori come parametri.

L'aggiunta del punto esclamativo impone una immediata interruzione dell'operazione in caso di errori di validazione, permettendo di evitare il salvataggio a database di valori scorretti per uno dei campi.

I metodi non specificano un tipo di ritorno; questo permette, in Rails, di stabilire una convenzione peculiare (non applicata nell'esempio): è possibile, anzi consigliato omettere il

return statement in quanto viene considerato come valore di ritorno l'ultima variabile menzionata al di fuori di una assegnazione.

```
    private

    def request_date_consistency
        if created_at?
            errors.add :request_date, :invalid unless request_date
               >= created_at.to_date
        else
            errors.add :request_date, :invalid unless request_date
               >= Date.today
        end
    end

    def shipment_date_consistency
        errors.add :shipment_date, :invalid unless shipment_date >
           request_date
    end
end
```

Listing 5.3: `shipment.rb`

Infine, nel blocco privato, vengono definiti i vincoli aggiuntivi per il *model*. L'attivazione del vincolo avviene con l'aggiunta di un errore a `errors`, che viene automaticamente registrata e segnalata da Rails. Notevole è il costrutto `unless`, che permette di evitare negazioni che offuscherebbero la chiarezza del codice.

5.2.2 Esempio di *controller*

In questa sezione verranno analizzati dei segmenti dell'implementazione di `API::SampleSessionController` che presentano caratteristiche implementative di interesse.

```
class API::SampleSessionController < ApiController
    before_action :api_authenticate
```

```
    before_action :set_sample_session, only: %i[show force_dispatch
        ...]
```

Listing 5.4: `sample_session_controller.rb`

La dichiarazione della classe è immediatamente seguita da due metodi definiti come *callback* da eseguire prima di ogni azione del controller, grazie alla *keyword* `before_action`. Prima di eseguire il metodo corrispondente ad una richiesta, verranno eseguiti i *callback* dichiarati in questo modo; essi generalmente si occupano di assegnare variabili globali o di autenticare la richiesta.

```
    def sync
        time_limit = DateTime.parse(Time.at(params[:
            last_synchronized].to_i).to_s)
        @sessions_to_sync = Session.where(sample_status: %w[ready
            dispatched analysed under_review final])
        @sessions_to_sync = @sessions_to_sync.where('updated_at >=
            :time_limit', time_limit: time_limit)
        render
    end

    def process_report
        @report = @sample_session.process_report_anonymized
        send_data @report.read, filename: "#{Time.now.strftime('%Y%
            m%d')}_#{params[:sample_code]}.pdf", type: 'application
            /pdf'
    ensure
        @report&.unlink
    end
```

Listing 5.5: `sample_session_controller.rb`

Quindi vengono definiti i metodi che rispondono alle richieste ricevute dal *controller*. Nell'esempio ne osserviamo due:

- Il primo, `sync`, effettua il *parsing* del timestamp fornito come parametro e lo usa per effettuare una *query* sulle istanze di `Session` da restituire. Quindi richiama la *view* corrispondente con il metodo `render`, che sarà in grado di ricavare automaticamente le risorse da rappresentare;

- Il secondo, `process_report`, richiede la generazione del referto dettagliato anonimo e ne effettua l'invio richiamando `send_data`, un metodo che permette l'invio di file come parte di una risposta HTTP.

Entrambi restituiscono una risposta con metodi differenti; è Rails a gestirle in modo da renderle visualizzabili dal client che ha effettuato la richiesta.

```
def api_authenticate
    @current_account = Account.find_by(access_id: ApiAuth.
        access_id(request))
    head(:unauthorized) unless @current_account && (ApiAuth.
        authentic?(request, @current_account.secret) || Rails.
        env == 'test')
end

# Use callbacks to share common setup or constraints between
    actions.
def set_sample_session
    @sample_session = Session.find_by(sample_code: params[:
        sample_code])
    return render(plain: "There is no session associated to
        sample code #{params[:sample_code]}.", status: :
        bad_request) unless @sample_session
    @sample_session
end

# Never trust parameters from the scary internet, only allow
    the white list through.
def sample_session_params
    params.permit(:sample_code, :reason, :report_notes, :
        gastroenterologist_notes_general, :
        gastroenterologist_notes_specific, :active_substance, :
        report, :timestamp)
end
```

Listing 5.6: **`sample_session_controller.rb`**

Nel blocco privato, alla fine, vengono implementati i *callback* da eseguire prima di ogni azione e viene definita la lista di parametri accettati: ciò permette di rendere più sicuro il *controller*. Il primo *callback* si occupa di autenticare la richiesta tramite la libreria ApiAuth, specializzata nell'aggiunta di autenticazione HMAC ad applicazioni Rails. Si tratta comunque di una funzionalità rimasta in sviluppo, qui riportata per puri scopi dimostrativi.

Il secondo *callback* effettua l'importante operazione di cercare l'istanza di `Session` indicata dal codice campione fornito come parametro e assegnarla come variabile globale per le operazioni del *controller*. Infine avviene la già menzionata definizione della *whitelist* dei parametri accettati.

I metodi del *controller* vengono così assegnati ai rispettivi *endpoint* nel file di *routing:*

```
get '/samples/sync', to: 'api/sample_session#sync'
get '/samples/:sample_code/report', to: 'api/sample_session#
    process_report'
```

Come si evince dal codice riportato, viene prima definito il metodo HTTP della richiesta, quindi viene specificato l'URI completo di eventuali parametri, ed infine il metodo del *controller* desiderato.

5.2.3 Esempio di *view*

Le *views* ricoprono un ruolo relativamente minore nell'applicativo; verrà quindi riportato un esempio corredato solo da una breve spiegazione.

```
json.sample_code sample_session.sample_code
json.sample_status sample_session.sample_status
json.questionnaire_answers do
  json.array! sample_session.questionnaire_answers do |answer|
    json.question answer.denormalized_question_text
    json.answer answer.get_answer
  end
end
```

Listing 5.7: `_data.jbuilder`

Si tratta di un frammento della *view* parziale usata per rappresentare una istanza di `Session`. Si può osservare come la struttura JSON sia costruita dichiarando il nome del campo per poi specificare il campo della risorsa che ne costituirà il valore. Nella costruzione del campo

`questionnaire_answers` viene usata una struttura iterativa per comporre un array contenente tutte le risposte al questionario presenti. Questa *view* viene elaborata in una struttura dati JSON direttamente da Rails tramite Jbuilder al momento opportuno.

5.3 Comunicazione

In questa sezione verranno presentati degli esempi di richieste e risposte per entrambi i tipi di API implementati.

5.3.1 REST

Richiesta

Risposta

5.3.2 GraphQL

Richiesta

Risposta

5.4 Immagini

You can prove anything you want by coldly logical reason—
if you pick the proper postulates.

Isaac Asimov

6 Verifica e validazione

L'applicativo sviluppato nel corso di questo progetto, come già menzionato, è entrato in fase di produzione (ovvero è stato rilasciato per l'uso da parte del cliente) relativamente presto, con aggiornamenti regolari che introducevano man mano nuove funzionalità. Tale situazione ha introdotto la forte necessità di controllare che ogni versione rilasciata del prodotto adempiesse correttamente a tutti i bisogni del cliente e fosse priva di inadeguatezze.

In questo capitolo i processi volti ad assicurare la qualità del prodotto, valutata termini di assenza di *failure* e adempimento ai bisogni espressi dal cliente.

6.1 Verifica

Il processo di verifica viene attuato durante lo sviluppo del prodotto con lo scopo di accertare che nuove funzionalità introdotte non siano causa di errori o malfunzionamenti. Tale processo deve quindi valutare sia il codice scritto, per rilevare eventuali errori di codifica, che il codice in esecuzione per verificare che si comporti secondo certe aspettative. Il processo di verifica viene effettuato applicando due tecniche: **analisi statica**, che adempie al primo compito, e **analisi dinamica** che permette di svolgere il secondo.

6.1.1 Analisi statica

La tecnica di analisi statica prevede di verificare il codice scritto al di fuori della sua esecuzione, effettuando dunque una revisione manuale volta ad evidenziare errori sintattici o altrimenti prontamente visibili allo sviluppatore. Essa permette di individuare subito eventuali imperfezioni che possono portare a comportamenti non previsti, se non a *fault* del sistema.

Nel corso del progetto, l'analisi statica del codice è stata eseguita costantemente durante lo sviluppo sotto la supervisione del tutor aziendale, permettendo di evitare non solo errori sintattici ma anche *bad practices* dovute all'inesperienza con il *framework* Rails.

L'analisi statica è stata supportata dall'utilizzo dell'IDE RubyMine, che mette a disposizione vari strumenti utili allo scopo ed è in grado di evidenziare alcune *bad practices* comuni, arrivando addirittura a segnalare metodi con eccessiva complessità ciclomatica, in congiunzione con la gemma RuboCop per assicurarsi di seguire la *style guide* di Ruby. RuboCop è stato anche utilizzato per automatizzare parte dell'analisi statica grazie alla sua funzione di autocorrezione, eseguibile sull'intero sorgente.

6.1.2 Analisi dinamica

La tecnica di analisi dinamica è complementare a quella di analisi statica, andando ad esaminare il comportamento del programma durante la sua esecuzione. Ciò avviene attraverso la definizione e l'esecuzione di test mirati a verificare il funzionamento del prodotto o di alcune sue componenti.

La realizzazione di una suite di test compare anche come requisito stabilito nella pianificazione dello stage (sezione 2.2.3.1, requisito D02). I test realizzati sono principalmente test di unità, mirati a verificare il comportamento di componenti specifiche del software, e di integrazione, per testare le interazioni tra tali componenti. I test di sistema non sono stati giudicati come rilevanti, risultando troppo complessi per essere implementati nei limiti temporali prefissati.

I test sono stati implementati utilizzando il *framework* RSpec, utilizzando varie gemme di supporto come Factory Bot, Shoulda Matchers e Database Cleaner. L'ambiente di *testing* è stato configurato in modo da effettuare una pulizia completa del database prima dell'esecuzione di ogni test e di ogni esempio parte di un test. I servizi di spedizione, *logging* e invio SMS sono rimpiazzati da classi *stub* create per simularne il comportamento. Le istanze dei *models* vengono create ad-hoc utilizzando Factory Bot prima dell'esecuzione di ogni gruppo di esempi.

I test sono stati eseguiti sia durante lo sviluppo che in presenza del tutor aziendale, per verificare il pieno soddisfacimento dei requisiti.

6.1.2.1 Test di unità

I test di unità realizzati vanno a verificare il corretto comportamento dei *models*. È stato realizzato un test per ciascun *model*, ognuno strutturato secondo le linee guida di RSpec: una prima suddivisione in scenari, che descrivono il contesto analizzato, a loro volta divisi in esempi, che implementano il caso specifico. La struttura di un test di unità è la seguente:

- **Scenario per i metodi di validazione**, i cui esempi testano i vincoli imposti sul *model*;
- **Scenario per le relazioni**, i cui esempi testano la corretta definizione delle relazioni con altri *models*; non fa parte dei test di integrazione, in quanto fa uso di *stub* e verifica semplicemente la corretta definizione dei vincoli di relazione;
- **Scenario per i callback**, i cui esempi testano i metodi *callback* definiti in vari contesti;
- **Scenario per i metodi di classe**, i cui esempi vanno a testare i metodi che esulano dalle categorie precedenti.

Gli esempi di ogni scenario sono raggruppati in contesti, i quali descrivono un particolare stato del sistema.

6.1.2.2 Test di integrazione

I test di integrazione realizzati riguardano principalmente il funzionamento della API GraphQL, che richiede appunto una perfetta integrazione tra i vari *models*. Questi test sono strutturati in modo differente, dato il loro diverso scopo: si ha un test per ogni *query*, che ne verifica i risultati in diversi contesti. Si compongono delle seguenti parti:

- **Query**, ovvero la definizione della query da testare in linguaggio GraphQL;
- **Autenticazione**, effettuata da un *callback* eseguito all'inizio di ogni test che fornisce i parametri di una sessione valida per fare la richiesta;
- **Contesti**, ovvero differenti situazioni ottenute istanziando i *models* coinvolti con vari attributi e relazioni, a cui appartengono gli esempi che verificano la correttezza della risposta alla *query* nel contesto dato.

6.2 Validazione

Il processo di validazione viene attuato al termine di una fase del progetto di sviluppo ed è mirato a valutare quanto il prodotto risultante rispecchi i bisogni e le esigenze del committente. La validazione eseguita sul prodotto dal tutor aziendale sia al termine dello sviluppo di ogni funzionalità che al termine dello stage ha dato un esito positivi, riscontrando il pieno soddisfacimento dei requisiti stabiliti in fase di pianificazione. Inoltre l'applicativo è entrato in produzione con successo già prima della fine dello stage, dimostrando la qualità della progettazione e dello sviluppo effettuati.

Era l'immagine di un compimento, e perciò qualcosa di assieme vitale e funebre.

Francesco Targhetta

7 Conclusione

Alla conclusione del progetto mi è stato possibile esprimere due tipi di valutazione sui risultati ottenuti: una oggettiva, basata su criteri misurabili quali il rispetto delle tempistiche e il soddisfacimento degli obiettivi, e una personale, data dall'impatto avuto dall'ambiente e dal metodo di lavoro nonché dalle conoscenze acquisite in corso di progetto.

Volendo fare una considerazione generale, l'esperienza ha avuto un'esito positivo: gli obiettivi fissati sono stati raggiunti senza grosse difficoltà, il prodotto realizzato è stato giudicato pronto ad essere utilizzato ed è stata espressa soddisfazione sul risultato del lavoro compiuto. Allo stesso modo ho avuto occasione di apprezzare un ambiente di lavoro accogliente e di grande aiuto alla mia crescita personale e professionale; e per questo voglio ringraziare lo staff di Moku.

7.1 Valutazione oggettiva

7.1.1 Raggiungimento degli obiettivi

Alla conclusione del periodo di stage, è possibile affermare che tutti gli obiettivi obbligatori e desiderabili posti in fase di pianificazione sono stati raggiunti con pieno successo. Per quanto riguarda gli obiettivi facoltativi, uno di essi è stato sviluppato ma non è stato rilasciato entro la fine dello stage (requisito Fo1, sezione 2.2.3.1, mentre ne sono stati implementati altri che portavano valore aggiunto a funzionalità già presenti.

Di seguito viene riportata una tabella che riassume gli obiettivi pianificati in riferimento alla sezione 2.2.3.1 ed il loro stato di raggiungimento al termine del lavoro:

Requisito	Stato
O01	Soddisfatto
O01	Soddisfatto
O03	Soddisfatto
D01	Soddisfatto
D02	Soddisfatto
D03	Soddisfatto
D04	Soddisfatto
F01	Soddisfatto, non rilasciato
F02	Soddisfatto

Tabella 7.1: Soddisfacimento degli obiettivi pianificati.

7.1.2 Consuntivo

Per quanto riguarda la pianificazione temporale, il limite di 300 ore è stato rispettato ed è risultato sufficiente, come esposto, per raggiungere gli obiettivi posti. Tuttavia, a causa di impegni accademici e alcuni giorni malattia, nom è bastata la settimana di *slack* prevista e il lavoro è stato distribuito anche nella settimana successiva, previa richiesta all'ufficio di Ateneo responsabile; le attività sono dunque giunte al termine il 17 luglio, contrariamente alla data prevista del 12 luglio (usando la settimana di *slack*). Ciò è evidentemente dovuto da una stima errata del tempo da dedicare agli impegni accademici concorrenti al periodo di stage, e l'eventuale dilazione dei termini era prevedibile.

Rispetto alla suddivisione oraria pianificata, sono state effettuate alcune modifiche in corso d'opera. Alcune attività, quali lo studio di nuove tecnologie e l'analisi dei requisiti, sono state "spalmate" nel periodo di stage in quanto necessarie dopo ogni *release* del prodotto per rivalutare le esigenze in collaborazione con il cliente. la preparazione dell'ambiente di sviluppo e la produzione di documentazione hanno richiesto meno tempo rispetto a quanto previsto grazie alle *best practices* in vigore nell'azienda, che ne hanno accelerati i tempi; le ore così risparmiate sono state assegnate alla progettazione e all'implementazione di test. La progettazione in particolare ha più che raddoppiato le ore previste, date le modifiche ai requisiti avvenute nelle prime settimane del progetto e la necessità di acquisire confidenza con la struttura dell'applicazione Rails. in compenso, una progettazione più dettagliata e ragio-

nata ha permesso di risparmiare tempo in fase implementativa, permettendo di codificare le funzionalità previste senza aggiunta di errori (ottenendo quindi correttezza per costruzione).

La tabella seguente riassume la ridistribuzione oraria al termine dello stage:

Attività	**Monte ore**
Comprensione sistema e obiettivi	20
Analisi dei requisiti	40
Progettazione	48 (+28)
Studio e setup ambiente di sviluppo	8 (-12)
Implementazione	134 (-16)
Test e validazione	40 (+10)
Documentazione	10 (-10)
Totale	300

Tabella 7.2: Totale di ore dedicato a ciascuna attività al termine dello stage.

7.2 Valutazione personale

7.2.1 Conoscenze acquisite

L'attività di stage volta mi ha permesso di apprendere e raffinare molte conoscenze, sia tecniche che professionali. Dal punto di vista tecnico, lavorare con un linguaggio di programmazione a me sconosciuto e con tecnologie molto differenti da quelle utilizzate in ambito accademico ha rappresentato una sfida molto interessante, che mi ha permesso di espandere i miei orizzonti in termini di paradigmi sia architetturali che implementativi, senza contare il valore aggiunto portato dall'avere appreso come sviluppare un'applicazione in Rails, un *framework* con grosse potenzialità. Ho inoltre imparato ad usare in modo più vantaggioso gli strumenti di sviluppo, in particolare le funzionalità di *debugging* di RubyMine, e di supporto: l'introduzione al paradigma *git-flow* è stata illuminante e lo utilizzerò sicuramente in progetti futuri.

L'interazione con il tutor aziendale, ma anche il resto del personale di Moku, è stata estremamente utile per acquisire esperienza sulle responsabilità e dui compiti di uno sviluppatore, nonché su come rapportarsi ad un cliente. Dalla capacità di affrontare difficoltà progettuali all'abilità di selezionare la tecnologia e l'approccio ottimali per soddisfare un requisito, sono molto soddisfatto delle conoscenze professionali acquisite durante questo progetto.

7.2.2 Crescita personale

Al termine dell'esperienza di stage, sono arrivato a considerarla com un importante passo anche nella mia crescita personale. Il contatto con l'ambiente lavorativo, le responsabilità assunte nel creare un prodotto che verrà utilizzato in un contesto reale e le sfide di *problem solving* che mi si sono presentate mi hanno permesso di ampliare i miei orizzonti e migliorarmi sia come persona che come futuro professionista. Moku mi ha accolto con un ambiente giovane, cordiale ed accogliente, in cui è stato possibile comunicare chiaramente e in cui le mie esigenze da studente sono state comprese. In conclusione, sono molto felice dell'esperienza fatta e la ritengo un passo molto importante della mia educazione.

A

Casi d'uso secondari

Di seguito vengono riportati i casi d'uso secondari per ogni caso d'uso principale, in riferimento alla sezione 3.2 del capitolo 3.

A.1 Casi d'uso secondari per UC1

A.1.1 UC1.1: Inserimento email

Attori: Utente non autenticato.
Scopo e descrizione: L'attore vuole inserire un indirizzo email per l'autenticazione al pannello di amministrazione.
Precondizioni: L'attore ha accesso alla pagina di autenticazione.
Scenario principale: L'attore inserisce un indirizzo mail nell'apposito campo.
Postcondizioni: L'attore ha inserito un indirizzo email valido.

A.1.2 UC1.2: Inserimento password

Attori: Utente non autenticato.
Scopo e descrizione: L'attore vuole inserire una password per l'autenticazione al pannello di amministrazione.
Precondizioni: L'attore ha accesso alla pagina di autenticazione.

Scenario principale: L'attore inserisce una password nel campo apposito.
Postcondizioni: L'attore ha inserito una password corretta.

A.2 Casi d'uso secondari per UC6

A.2.1 UC6.1: Riordinamento tabella dei record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole cambiare l'ordine della tabella dei record secondo un certo campo.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale: L'attore seleziona la colonna del campo in base al quale ordinare la tabella. **Postcondizioni:** L'attore ha riordinato la tabella secondo il campo desiderato.

A.2.2 UC6.2: Applicazione filtro alla tabella dei record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole filtrare alcune righe della tabella dei record.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale:

- L'attore seleziona l'icona del filtro;
- L'attore imposta il filtro desiderato;
- L'attore clicca sul pulsante"Filtra".

Postcondizioni: L'attore ha applicato il filtro desiderato alla tabella dei record.

A.2.3 UC6.3: Rimozione filtro alla tabella dei record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole rimuovere un filtro dalla tabella dei record.
Precondizioni: L'attore sta visualizzando una tabella di record con un filtro attivo.
Scenario principale:

- L'attore seleziona l'icona del filtro;

- L'attore clicca sul pulsante "Rimuovi filtri".

Postcondizioni: L'attore ha rimosso il filtro desiderato alla tabella dei record.

A.2.4 UC6.4: Aggiunta di un nuovo record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole aggiungere un record alla tabella.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale:

- L'attore clicca sul pulsante "Aggiungi **{record}**", dove **{record}** è il nome della categoria selezionata;
- L'attore viene reindirizzato al *form* di creazione del record;
- L'attore riempie gli appositi campi con i dati desiderati;
- L'attore clicca sul pulsante "Crea **{record}**".

Postcondizioni: L'attore ha creato con successo un nuovo record.
Estensioni: L'attore annulla la creazione di un nuovo record (UC6.9).

A.2.5 UC6.5: Visualizzazione di un record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole visualizzare un record della tabella.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale:

- L'attore clicca sul pulsante "Mostra" sulla riga corrispondente al record desiderato;
- L'attore visualizza i dati relativi al record;
- L'attore modifica il record;
- L'attore elimina il record.

Postcondizioni: L'attore ha eseguito tutte le operazioni di gestione della categoria di record selezionata con successo.

A.2.6 UC6.6: Modifica di un record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole modificare un record della tabella.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale:

- L'attore clicca sul pulsante "Modifica" sulla riga corrispondente al record desiderato;
- L'attore viene reindirizzato al *form* di modifica del record;
- L'attore modifica il contenuto dei campi disponibili;
- L'attore clicca sul pulsante "Aggiorna **{record}**", dove **{record}** è il nome della categoria selezionata.

Scenario alternativo:

- L'attore visualizza il record desiderato (UC6.5);
- L'attore clicca sul pulsante "Modifica **{record}**", dove **{record}** è il nome della categoria selezionata;
- L'attore viene reindirizzato al *form* di modifica del record;
- L'attore modifica il contenuto dei campi disponibili;
- L'attore clicca sul pulsante "Aggiorna **{record}**";

Postcondizioni: L'attore ha correttamente aggiornato il record con i dati desiderati.
Estensioni:

- L'attore annulla la modifica del record (UC6.9);
- L'attore commette un errore nella modifica del record (UC6.8).

A.2.7 UC6.7: Eliminazione di un record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole eliminare un record della tabella.
Precondizioni: L'attore sta visualizzando una tabella di record.
Scenario principale:

- L'attore clicca sul pulsante "Rimuovi" sulla riga corrispondente al record desiderato;
- L'attore visualizza un *pop-up* di conferma dell'operazione;
- L'attore clicca sul pulsante "Ok" per confermare l'operazione.

Scenario alternativo:

- L'attore visualizza il record desiderato (UC6.5);
- L'attore clicca sul pulsante "Rimuovi **{record}**", dove **{record}** è il nome della categoria selezionata;
- L'attore visualizza un *pop-up* di conferma dell'operazione;
- L'attore clicca sul pulsante "Ok" per confermare l'operazione.

Postcondizioni: L'attore ha correttamente rimosso il record.
Estensioni: L'attore annulla la rimozione del record (UC6.9).

A.2.8 UC6.8: Errore nella modifica di un record

Attori: Amministratore.
Scopo e descrizione: L'attore vuole modificare un record.
Precondizioni: L'attore sta visualizzando il *form* di modifica del record.
Scenario principale:

- L'attore inserisce uno o più valori non validi in uno o più dei campi disponibili;
- L'attore visualizza un messaggio di errore, il quale descrive il valore invalido inserito e l'errore riscontrato.

Postcondizioni: Il messaggio di errore rimane in cima al *form* di modifica e la modifica viene prevenuta.

A.2.9 UC6.9: Annullamento operazione

Attori: Amministratore.
Scopo e descrizione: L'attore vuole annullare l'operazione in corso.
Precondizioni: L'attore non ha ancora terminato l'operazione precedentemente selezionata.
Scenario principale: L'attore clicca sul pulsante "Annulla".
Postcondizioni: L'attore ha annullato correttamente l'operazione ed eventuali modifiche vengono prevenute.

A.2.10 UC6.10: Gestione di un utente

Attori: Amministratore.
Scopo e descrizione: L'attore vuole gestire un utente autorizzato.
Precondizioni: L'attore ha seleziona la scheda "Utenti".
Scenario principale:

- L'attore visualizza il record corrispondente all'utente desiderato (UC6.5);
- L'attore genera una nuova password per l'utente (UC6.10.1).

Postcondizioni: L'attore ha eseguito tutte le operazioni di gestione dell'utente desiderato con successo.

A.2.11 UC6.10.1: Generazione nuova password per un utente

Attori: Amministratore.
Scopo e descrizione: L'attore vuole generare una nuova password per un utente autorizzato.
Precondizioni: L'attore sta visualizzando il record relativo all'utente.
Scenario principale: L'attore preme sul pulsante "Rigenera password".
Postcondizioni: L'attore ha correttamente generato una nuova password per l'utente desiderato.

A.2.12 UC6.11: Gestione di un cliente

Attori: Amministratore.
Scopo e descrizione: L'attore vuole gestire un cliente.
Precondizioni: L'attore ha seleziona la scheda "Clienti".
Scenario principale:

- L'attore visualizza il record corrispondente all'utente desiderato (UC6.5);
- L'attore scarica il modulo di consenso al trattamento dei dati (UC6.11.1).

Postcondizioni: L'attore ha eseguito tutte le operazioni di gestione del cliente desiderato con successo.

A.2.13 UC6.11.1: Scaricamento modulo di consenso al trattamento dei dati

Attori: Amministratore.
Scopo e descrizione: L'attore vuole scaricare il modulo per il consenso al trattamento dei dati firmato da un cliente (se presente).
Precondizioni: L'attore sta visualizzando il record relativo al cliente desiderato.
Scenario principale: L'attore preme sul pulsante "Scarica privacy policy".
Postcondizioni: L'attore ha correttamente scaricato il modulo (se presente).

A.3 Casi d'uso secondari per UC11

A.3.1 UC11.1: Richiesta informazioni lista degli utenti

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere la lista degli utenti con ruolo "pharmacy" per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati degli utenti appartenenti alla farmacia corrente.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.2 UC11.2: Richiesta informazioni utente

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un utente con ruolo "pharmacy" per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati dell'utente desiderato fornendone il codice identificativo interno.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.3 UC11.3: Richiesta reset della password di un utente

Attori: Front end.
Scopo e descrizione: L'attore vuole richiedere il reset della password per un utente con ruolo "pharmacy".
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore richiede al sistema il reset della password dell'utente desiderato fornendone il codice identificativo interno.
- Il sistema invia un'email per la procedura di reset all'indirizzo mail dell'utente specificato.

Postcondizioni: La richiesta di reset viene processata correttamente dal sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.4 UC11.4: Impostazione nuova password per un utente

Attori: Front end.
Scopo e descrizione: L'attore vuole impostare.
Precondizioni: L'attore è collegato al sistema e possiede un *token* valido per la reimpostazione della password per l'utente autorizzato desiderato.
Scenario principale: L'attore invia al sistema la richiesta di reimpostazione contenente

il *token* di reset e la nuova password desiderata.
Postcondizioni: Il sistema ha reimpostato correttamente la password per l'utente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.5 UC11.5: Creazione utente

Attori: Front end.
Scopo e descrizione: L'attore vuole creare un utente con ruolo "pharmacy" per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di creazione di un utente contenente un indirizzo email valido e una password.
Postcondizioni: Il sistema ha creato correttamente l'utente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.6 UC11.6: Rimozione utente

Attori: Front end.
Scopo e descrizione: L'attore vuole eliminare un utente con ruolo "pharmacy" per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema di eliminare l'utente desiderato fornendone il codice identificativo interno.
Postcondizioni: Il sistema ha rimosso correttamente l'utente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.3.7 UC11.7: Aggiornamento dati utente

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare i dati di un utente con ruolo "pharmacy"

per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di aggiornamento dei dati dell'utente desiderato fornendone il codice identificativo interno e i nuovi dati da inserire.
Postcondizioni: Il sistema ha aggiornato correttamente l'utente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4 Casi d'uso secondari per UC12

A.4.1 UC12.1: Richiesta storico degli esami

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere lo storico degli esami per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati degli esami effettuati dalla farmacia corrente.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.2 UC12.2: Richiesta storico degli esami in base all'avanzamento

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere lo storico degli esami per la farmacia corrente che si trovino in un certo stato di avanzamento.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore richiede al sistema i dati dell'esame desiderato fornendone lo stato di avanzamento desiderato;

- Il sistema filtra gli esami in base allo stato specificato e invia una risposta contenente i dati relativi agli esami filtrati.

Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.3 UC12.3: Richiesta informazioni esame

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un esame effettuato presso la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati dell'esame desiderato fornendone il codice identificativo interno.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.4 UC12.4: Richiesta informazioni esame in base al campione

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un esame effettuato presso la farmacia corrente con uno specifico codice campione.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale:

- L'attore richiede al sistema i dati dell'esame desiderato fornendone il codice identificativo del campione associato;

- Il sistema ricerca l'esame desiderato e invia una risposta contenente i relativi dati.

Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.5 UC12.5: Creazione esame

Attori: Front end.
Scopo e descrizione: L'attore vuole creare un esame presso la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di creazione di un esame contenente il codice identificativo interno del cliente associato e il codice del campione assegnato.
Postcondizioni: Il sistema ah creato correttamente l'esame e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.6 UC12.6: Rimozione esame

Attori: Front end.
Scopo e descrizione: L'attore vuole eliminare un esame creato presso la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema di eliminare l'esame desiderato fornendone il codice identificativo interno.
Postcondizioni: Il sistema ha rimosso correttamente l'esame desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.7 UC12.7: Aggiornamento dati esame

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare i dati di un esame effettuato presso la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di aggiornamento dei dati dell'esame desiderato fornendone il codice identificativo interno e i nuovi dati da inserire.
Postcondizioni: Il sistema ha aggiornato correttamente l'esame desiderato e l'attore ha

ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.4.8 UC12.8: Aggiornamento risposte al questionario di un esame

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare le risposte date al questionario di un esame.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di aggiornamento delle risposte al questionario dell'esame desiderato fornendone il codice identificativo interno e le risposte da inserire.
Postcondizioni: Il sistema ha aggiornato correttamente le risposte al questionario associato all'esame desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5 Casi d'uso secondari per UC13

A.5.1 UC13.1: Richiesta informazioni registro dei clienti

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere il registro dei clienti della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati dei clienti appartenenti alla farmacia corrente.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5.2 UC13.2: Richiesta informazioni cliente

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di un cliente della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati di un cliente appartenente alla farmacia corrente fornendone il codice identificativo interno.
Postcondizioni: L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5.3 UC13.3: Creazione cliente

Attori: Front end.
Scopo e descrizione: L'attore vuole creare un cliente per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di creazione di un cliente contenente i dati necessari.
Postcondizioni: Il sistema ha creato correttamente il cliente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5.4 UC13.4: Rimozione cliente

Attori: Front end.
Scopo e descrizione: L'attore vuole eliminare un cliente per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema di eliminare il cliente desiderato fornendone il codice identificativo interno.
Postcondizioni: Il sistema ha rimosso correttamente il cliente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5.5 UC13.5: Aggiornamento dati cliente

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare i dati di un cliente della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di aggiornamento dei dati del cliente desiderato fornendone il codice identificativo interno e i nuovi dati da inserire.
Postcondizioni: Il sistema ha aggiornato correttamente il cliente desiderato e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.5.6 UC13.6: Invio codice OTP

Attori: Front end.
Scopo e descrizione: L'attore vuole inviare il codice per l'accettazione della privacy policy ad un cliente della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy", il cliente possiede un numero di cellulare valido.
Scenario principale:

- L'attore richiede al sistema l'invio del codice al cliente desiderato fornendone il codice identificativo interno e i nuovi dati da inserire;
- Il sistema richiede l'invio di un SMS con il codice desiderato ad Amazon SNS.

Postcondizioni: Il sistema ha inoltrato correttamente la richiesta di invio e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni:

- L'attore invia una richiesta non valida (UC16);
- L'attore effettua troppi tentativi di invio (UC17).

A.5.7 UC13.7: Verifica codice OTP

Attori: Front end.
Scopo e descrizione: L'attore vuole verificare il codice per l'accettazione della privacy policy da parte di un cliente della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy", il cliente ha ricevuto il codice OTP.
Scenario principale:

- L'attore richiede al sistema la verifica del codice del cliente desiderato fornendo il codice OTP e il codice identificativo interno del cliente.

- Il sistema verifica la validità del codice.

Postcondizioni: Il sistema ha registrato l'accettazione della privacy policy, e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia un codice non valido (UC16).

A.5.8 UC13.8: Caricamento manuale della *privacy policy*

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiungere al record di un cliente della farmacia una copia del modulo per il consenso al trattamento dei dati, firmato dal cliente .
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy", esiste una copia digitale del modulo firmato.
Scenario principale: L'attore invia al sistema una richiesta di registrazione dell'accettazione della privacy policy, fornendo la copia digitale del modulo e il codice identificativo interno del cliente.
Postcondizioni: Il sistema ha registrato correttamente l'accettazione della *privacy policy*, ha caricato correttamente la copia del modulo e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.6 Casi d'uso secondari per UC14

A.6.1 UC14.1: Richiesta informazioni lista delle spedizioni

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere la lista delle spedizioni per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati delle spedizioni appartenenti alla farmacia corrente. **Postcondizioni:** L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.6.2 UC14.2: Richiesta informazioni spedizione

Attori: Front end.
Scopo e descrizione: L'attore vuole ottenere i dati di una spedizione per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema i dati di una spedizione appartenente alla farmacia corrente fornendone il codice identificativo interno. **Postcondizioni:** L'attore ha ricevuto correttamente i dati richiesti al sistema.
Estensioni: L'attore invia una richiesta non valida (UC16).

A.6.3 UC14.3: Creazione spedizione

Attori: Front end.
Scopo e descrizione: L'attore vuole creare una spedizione per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di creazione di una spedizione contenente i dati necessari e una lista di campioni da spedire. **Postcondizioni:** Il

sistema ha creato correttamente la spedizione desiderata e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione. **Estensioni:** L'attore invia una richiesta non valida (UC16).

A.6.4 UC14.4: Rimozione spedizione

Attori: Front end.
Scopo e descrizione: L'attore vuole eliminare una spedizione per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore richiede al sistema di eliminare la spedizione desiderata fornendone il codice identificativo interno. **Postcondizioni:** Il sistema ha rimosso correttamente la spedizione desiderata e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione. **Estensioni:** L'attore invia una richiesta non valida (UC16).

A.6.5 UC14.5: Aggiornamento dati spedizione

Attori: Front end.
Scopo e descrizione: L'attore vuole aggiornare i dati di un cliente della farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo "pharmacy".
Scenario principale: L'attore invia al sistema una richiesta di aggiornamento dei dati della spedizione desiderata fornendone il codice identificativo interno e i nuovi dati da inserire. **Postcondizioni:** Il sistema ha aggiornato correttamente la spedizione desiderata e l'attore ha ricevuto un messaggio di conferma del successo dell'operazione. **Estensioni:** L'attore invia una richiesta non valida (UC16).

A.6.6 UC14.6: Visualizzazione lettera di vettura

Attori: Front end.
Scopo e descrizione: L'attore vuole visualizzare la lettera di vettura di una spedizione per la farmacia corrente.
Precondizioni: L'attore è collegato al sistema e possiede una sessione attiva per un utente

con ruolo “pharmacy”, i dati della spedizione sono validi.
Scenario principale:

- L’attore richiede al sistema la lettera di vettura per la spedizione desiderata fornendone il codice identificativo interno;
- Il sistema ottiene la lettera di vettura dalle API SDA;
- Il sistema invia la lettera di vettura all’attore.

Postcondizioni: L’attore visualizza correttamente la lettera di vettura.
Estensioni: L’attore invia una richiesta non valida (UC16).

A.6.7 UC14.7: Prenotazione spedizione

Attori: Front end.
Scopo e descrizione: L’attore vuole prenotare una spedizione per la farmacia corrente presso SDA.
Precondizioni: L’attore è collegato al sistema e possiede una sessione attiva per un utente con ruolo “pharmacy”, i dati della spedizione sono validi.
Scenario principale:

- L’attore richiede al sistema la prenotazione della spedizione desiderata fornendone il codice identificativo interno;
- Il sistema effettua la prenotazione attraverso le API SDA.

Postcondizioni: Il sistema ha registrato il codice di prenotazione della spedizione e la data di ritiro, e l’attore ha ricevuto i dati aggiornati della spedizione e un messaggio di conferma del successo dell’operazione.
Estensioni: L’attore invia una richiesta non valida (UC16);

B

Tracciamento tra requisiti e casi d'uso

B.1 Tracciamento requisiti-casi d'uso

Sigla	Descrizione
RF1	UC1
RF1.1	UC1.1
RF1.2	UC1.2
RF1.3	UC2
RF2	UC3
RF2.1	UC5
RF3	UC4
RF4	UC6
RF4.1	UC6.1
RF4.2	UC6.2
RF4.3	UC6.3
RF4.4	UC6.4
RF4.5	UC6.5

RF4.6	UC6.6
RF4.7	UC6.7
RF4.8	UC6.10
RF4.8.1	UC6.10.1
RF4.9	UC6.11
RF4.9.1	UC6.11.1
RF4.10	UC69
RF4.11	UC68
RF5	UC77
RF6	UC8
RF7	UC9
RF8	UC10
RF9	UC11
RF9.1	UC11.1
RF9.2	UC11.2
RF9.3	UC11.3
RF9.4	UC11.4
RF9.5	UC11.5
RF9.6	UC11.6
RF9.7	UC11.7
RF10	UC12
RF10.1	UC12.1
RF10.1.1	UC12.2
RF10.2	UC12.3
RF10.2.1	UC12.4
RF10.3	UC12.5
RF10.4	UC12.6
RF10.5	UC12.7

RF10.5.1	UC12.8
RF11	UC13
RF11.1	UC13.1
RF11.2	UC13.2
RF11.3	UC13.3
RF11.4	UC13.4
RF11.5	UC13.5
RF11.6	UC13.6
RF11.6.1	UC17
RF11.7	UC13.7
RF11.8	UC13.8
RF12	UC14
RF12.1	UC14.1
RF12.2	UC14.2
RF12.3	UC14.3
RF12.4	UC14.4
RF12.5	UC14.5
RF12.6	UC14.6
RF12.7	UC14.7
RF13	UC15
RF14	UC16
RF15	UC18
RF16	UC19
RF17	UC20
RF18	UC21
RF19	UC22
RF20	UC23
RF21	UC24

RF22	UC25
RF23	UC26
RF24	UC27
RF25	UC28
RF26	UC29
RF27	UC30
RF28	UC31
RF29	UC32
RF30	UC33
RF31	UC34

Tabella B.1: Tracciamento requisiti-casi d'uso.

B.2 Tracciamento casi d'uso-requisiti

Sigla	Descrizione
UC1	RF1
UC1.1	RF1.1
UC1.2	RF1.2
UC2	RF1.3
UC3	RF2
UC5	RF2.1
UC4	RF3
UC6	RF4
UC6.1	RF4.1
UC6.2	RF4.2
UC6.3	RF4.3
UC6.4	RF4.4
UC6.5	RF4.5

UC6.6	RF4.6
UC6.7	RF4.7
UC68	RF4.11
UC69	RF4.10
UC6.10	RF4.8
UC6.10.1	RF4.8.1
UC6.11	RF4.9
UC6.11.1	RF4.9.1
UC77	RF5
UC8	RF6
UC9	RF7
UC10	RF8
UC11	RF9
UC11.1	RF9.1
UC11.2	RF9.2
UC11.3	RF9.3
UC11.4	RF9.4
UC11.5	RF9.5
UC11.6	RF9.6
UC11.7	RF9.7
UC12	RF10
UC12.1	RF10.1
UC12.2	RF10.1.1
UC12.3	RF10.2
UC12.4	RF10.2.1
UC12.5	RF10.3
UC12.6	RF10.4
UC12.7	RF10.5

UC12.8	RF10.5.1
UC13	RF11
UC13.1	RF11.1
UC13.2	RF11.2
UC13.3	RF11.3
UC13.4	RF11.4
UC13.5	RF11.5
UC13.6	RF11.6
UC13.7	RF11.7
UC13.8	RF11.8
UC14	RF12
UC14.1	RF12.1
UC14.2	RF12.2
UC14.3	RF12.3
UC14.4	RF12.4
UC14.5	RF12.5
UC14.6	RF12.6
UC14.7	RF12.7
UC15	RF13
UC16	RF14
UC17	RF11.6.1
UC18	RF15
UC19	RF16
UC20	RF17
UC21	RF18
UC22	RF19
UC23	RF20
UC24	RF21

UC25	RF22
UC26	RF23
UC27	RF24
UC28	RF25
UC29	RF26
UC30	RF27
UC31	RF28
UC32	RF29
UC33	RF30
UC34	RF31

Tabella B.2: Tracciamento casi d'uso-requisiti.

Ringraziamenti

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi commodo, ipsum sed pharetra gravida, orci magna rhoncus neque, id pulvinar odio lorem non turpis. Nullam sit amet enim. Suspendisse id velit vitae ligula volutpat condimentum. Aliquam erat volutpat. Sed quis velit. Nulla facilisi. Nulla libero. Vivamus pharetra posuere sapien. Nam consectetuer. Sed aliquam, nunc eget euismod ullamcorper, lectus nunc ullamcorper orci, fermentum bibendum enim nibh eget ipsum. Donec porttitor ligula eu dolor. Maecenas vitae nulla consequat libero cursus venenatis. Nam magna enim, accumsan eu, blandit sed, blandit a, eros.