Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Analisi, progettazione e sviluppo del backend di un'applicazione web per la gestione di eventi

Tesi di laurea

Relatore
Prof. Davide Bresolin

Laureando
Alberto Lazari

Anno Accademico 2021-2022

Sommario

La tesi descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, presso la sede di Treviso di Moku S.r.l., il cui obiettivo era la reimplementazione del backend di una piattaforma di gestione di eventi, sfruttando gli strumenti tipicamente utilizzati nei progetti dell'azienda.

In particolare, i seguenti capitoli tratteranno del contesto lavorativo dell'azienda, dell'analisi svolta sullo stato della piattaforma ad inizio stage, della progettazione e successiva implementazione iniziale del nuovo backend, focalizzando l'attenzione sulle scelte stilistiche e architetturali perseguite.

Ringraziamenti

Voglio ringraziare il Prof. Davide Bresolin, per l'interesse, il supporto e l'aiuto fornito durante il periodo di stage e di stesura di questa tesi.

Ringrazio i miei genitori e tutti i miei famigliari per il supporto e l'affetto che mi hanno donato durante questi hanno di studio.

Ringrazio i colleghi di Moku che mi hanno accolto calorosamente tra loro durante la mia esperienza di stage, in particolare Riccardo e Nicolò, per avermi sempre fornito l'aiuto che cercavo durante il mio lavoro.

Ringrazio la Comunità Capi del Mestre 2, per avermi accompagnato fin qui, infondendo in me una maturità e una consapevolezza che mi hanno permesso di raggiungere questo traguardo.

Ringrazio i miei colleghi studenti, stagisti e gli amici dei gruppi di progetto, con cui ho condiviso le difficoltà e le soddisfazioni di quest'anno.

Infine ringrazio i miei amici, che mi sono sempre stati vicini e con cui ho condiviso esperienze indimenticabili.

Padova, Luglio 2022 Alberto Lazari

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli esempi di codice

Capitolo 1

L'azienda

1.1 Descrizione generale

Moku S.r.l. è una start-up nata nel 2013 all'interno di un progetto supportato da H-Farm. Dopo aver abbandonato il progetto si è dedicata allo sviluppo software su commissione e consulenza, per poi allontanarsi definitivamente da H-Farm a settembre 2021, muovendo la sua sede dalla *farm* a Roncade a quella attuale di Treviso.

L'azienda è in continua espansione e conta circa 20 dipendenti, la maggior parte con età inferiore ai 30 anni. Questo contribuisce a mantenere l'ambiente di lavoro stimolante e accogliente per tutti, permettendo di includere i diversi studenti che ogni anno svolgono il loro stage presso l'azienda. Per questi vengono attivate proposte di progetto per i ruoli di sviluppatore backend, sviluppatore frontend e sviluppatore mobile, all'interno di team interni che lavorano a progetti reali commissionati all'azienda.

1.2 Modello di sviluppo

I progetti di Moku seguono un modello di sviluppo *agile*, con metodologie basate su *Scrum*[1], un framework pensato per team di sviluppo software di piccole dimensioni (non più di dieci membri). La metodologia adottata prevede le seguenti caratteristiche:

- il lavoro viene suddiviso in *sprint*, intervalli temporali della durata di due settimane;

[1] Jeff Sutherland Ken Schwaber. *The 2020 Scrum Guide*. URL: https://scrumguides.org/scrum-guide.html.

- ogni *sprint* è preceduto da una riunione di pianificazione degli obiettivi, espressi sotto forma di *user stories*, che esprimono le funzionalità del software da implementare, scritte in un linguaggio naturale dal punto di vista dell'utente;
- uno *sprint* termina con la relativa *sprint review*, una riunione con il cliente che ha l'obiettivo di mostrare l'incremento prodotto nel software, attraverso dimostrazioni del funzionamento del software stesso;
- il team a cui viene affidato lo sviluppo di un progetto è composto da diverse figure professionali, tra cui un *project manager*, il cui compito è coordinare il lavoro tra gli altri componenti e definire le *user stories* da inserire nel *backlog* degli *sprint*, oltre a gestire la pianificazione dello *sprint* stesso;
- all'inizio di ogni giornata lavorativa, il team si riunisce nello *stand-up meeting*, una riunione della durata di circa 15 minuti, per condividere lo stato del lavoro di ogni componente, descrivere gli obiettivi del giorno e far emergere eventuali problemi sorti durante lo sviluppo.

La metodologia adottata permette di avere una comunicazione regolare ed efficace tra il team di sviluppo e il cliente, che porta a una definizione più semplice e precisa dei requisiti che il prodotto deve rispettare e a una comprensione immediata e chiara dell'avanzamento dello sviluppo da parte del cliente, attraverso le dimostrazioni pratiche effettuate nel contesto delle *sprint review*.

Capitolo 2

Descrizione dello stage

2.1 Introduzione al progetto

2.1.1 Presentazione del dominio

Infinite Area, una piattaforma dell'innovazione, ha ideato e sviluppato un'applicazione web per l'organizzazione e la gestione di eventi e conferenze dal vivo, online e ibride, denominata Evvvents. L'applicazione si propone come un *software as a service*, configurabile nell'aspetto e nelle funzionalità, in base alle preferenze dell'utente. In questo modo ogni utente che possiede un piano di iscrizione all'applicazione, un *tenant*, gestisce la sua versione del software, rendendo Evvvents nel suo complesso un *software multi tenant*. I *tenant* si identificano all'interno dell'applicazione come "piattaforme"; ogni piattaforma gestisce più aziende, dette "organizzatori", perché dirette responsabili della creazione e organizzazione degli eventi, il fulcro del software.

L'applicazione web è pensata per gestire l'organizzazione degli eventi. Non viene utilizzata dagli utenti finali, cioè quelli che si iscrivono e partecipano agli eventi, perché loro utilizzano un'applicazione *mobile*. Ci sono diversi ruoli di utenti in tutto il sistema, che hanno permessi e funzioni diverse:

- super admin: ha una visione ad alto livello dell'applicazione e può gestirne ogni aspetto. Si occupa principalmente di creare le piattaforme e i loro admin;
- admin: gestisce tutto all'interno della sua piattaforma, in particolare gli organizzatori;
- manager: si occupa di gestire l'organizzatore (l'azienda all'interno della piattaforma) e di pianificare gli eventi;
- staff: gestisce principalmente i check-in dei partecipanti durante gli eventi. Si occupa della gestione dei singoli eventi che gli vengono assegnati;
- *speaker* o relatore: partecipa agli eventi per fare degli interventi;
- partecipante: può essere iscritto agli eventi. Non può accedere o eseguire nessun tipo di azione dall'interfaccia web.

Per quanto riguarda gli eventi, sono presenti diverse funzionalità disponibili ai partecipanti e agli altri tipi di utenti:

Sistemi di videoconferenze Per ogni evento è possibile specificare l'integrazione con diversi sistemi di videoconferenze, tra cui:

- Zoom meeting e webinar
- Microsoft Teams
- GoToWebinar
- Webex
- Sistemi esterni di videoconferenze

Sistema di check-in Nelle pagine degli eventi vengono riportati i check-in effettuati in presenza e in remoto, per tenere traccia del numero di utenti partecipanti.

Comunicazione tra utenti È presente un sistema di messaggistica tra i partecipanti, online e in presenza. I partecipanti possono anche avere delle conversazioni con i relatori dell'evento.

Notifiche Ogni piattaforma può decidere di richiedere l'integrazione con il sistema di notifiche push e SMS per gli eventi. Le notifiche possono essere anche inviate via mail ai partecipanti e possono essere automatiche o programmate.

Sistema di crediti Le piattaforme possono anche decidere di utilizzare l'integrazione con il sistema dei crediti, in cui ogni evento permette agli utenti di guadagnare un numero specificato di crediti, in base alla durata della partecipazione all'evento.

2.1.2 Il progetto

Dopo aver realizzato una prima versione prototipale di Evvvents, Infinite Area si è rivolta a Moku, chiedendo di apportare modifiche e miglioramenti all'applicazione. A seguito di un'attività di analisi effettuata da Moku, è risultato che il frontend sarebbe stato mantenuto e fatto evolvere, mentre il backend presentava grossi limiti, funzionali e strutturali. Mantenerlo e modificarlo per raggiungere uno standard di qualità adeguato sarebbe costato più tempo e risorse di una riscrittura completa, quindi è stata presa la decisione di svilupparlo da zero con le tecnologie utilizzate da Moku per prassi aziendale, per quanto possibile, puntando a ricreare una API quanto più simile a quella esistente, per limitare le modifiche necessarie sul frontend.

Lo scopo del progetto di stage è stata la realizzazione della nuova versione del backend, partendo da un'attività di analisi dettagliata della versione originale, passando per la progettazione della nuova versione.

2.2 Requisiti

I requisiti dello stage riportati nel piano di lavoro sono i seguenti, categorizzati per importanza:

Obbligatori Requisiti primari, necessari per una buona riuscita dello stage:

- gestione e pianificazione del progetto attraverso kanban board condivisa;
- analisi dei flussi attuali e delle API richieste;
- progettazione ed implementazione dei modelli e dei controller, a partire dai requisiti raccolti;
- analisi ed integrazione Zoom, GoToWebinar, Webex.

Desiderabili Non necessari, ma che contribuiscono alla completezza del prodotto, se rispettati:

- coordinamento con il cliente finale;
- integrazione team;
- integrazione stampante biglietti;
- suite di testing del software prodotto;
- documentazione completa.

Opzionali Che portano del valore aggiunto al progetto:

- ulteriori modifiche all'applicazione che esulano da quando riportato nel piano di lavoro.

Il conseguimento dei requisiti è stato in parte dipendente dalle decisioni di gestione del progetto da parte del *project manager* e dalle richieste o dalla disponibilità del committente. In particolare è stato chiesto di dedicare una quantità limitata di tempo al testing, perché non richiesto esplicitamente e non è stato possibile implementare le integrazioni, a causa di ritardi nelle risposte del committente, necessarie per definire alcuni aspetti.

2.3 Pianificazione

Il periodo di svolgimento dello stage era previsto tra il 26 aprile 2022 e il 1 luglio 2022, per una durata complessiva di 300 ore. Il periodo preventivato considera due settimane aggiuntive a quelle necessarie a raggiungere 300 ore, utili a coprire eventuali imprevisti. La tabella che segue mostra la pianificazione delle attività da svolgere per ogni settimana, considerando 8 ore di lavoro al giorno:

2.4 Tecnologie utilizzate

Essendo il progetto un lavoro di riscrittura completa, è stato deciso di sfruttare quasi tutte le tecnologie comunemente utilizzate per convenzione aziendale nello sviluppo di backend:

Settimana	Attività
1 27/04 - 29/04	Comprensione sistema e obiettivi Analisi dei requisiti
2 02/05 - 06/05	Progettazione
3 09/05 - 13/05	Progettazione Studio e setup ambiente di sviluppo
4 16/05 - 20/05	Implementazione
5 23/05 - 27/05	Implementazione
6 30/05 - 03/06	Implementazione
7 06/06 - 10/06	Implementazione Test e validazione
8 13/06 - 17/06	Test e validazione Documentazione

Tabella 2.1: Tabella della pianificazione del lavoro

2.4.1 Ruby

Ruby[1] è un linguaggio di *scripting* interpretato, a oggetti e dinamicamente tipizzato, che punta alla produttività, mantenendo una sintassi semplice ed elegante e una bassa curva di apprendimento. La sintassi prevede un uso limitato della punteggiatura e di elementi considerati superflui, preferendo uno stile spesso naturale e intuitivo, che migliora la leggibilità del codice:

```
user.updated_at = 1.hour.ago
```

Codice 1: Esempio della sintassi di Ruby

In Ruby ogni *statement* è un'espressione, che ritorna un risultato, infatti ogni blocco di codice ritorna il risultato dell'ultima istruzione. Nell'esempio seguente la funzione `get_image_from` ritorna l'immagine dall'array `attribute`, se presente; altrimenti ritorna `nil`, perché l'espressione non ha ritornato un risultato:

```
def get_image_from attribute do
  attribute[:image] if attribute.has_key? :image
end

user.image = get_image_from uploaded_files
```

Codice 2: Esempio di una funzione in Ruby

[1] *Sito ufficiale di Ruby.* URL: https://www.ruby-lang.org/.

Il linguaggio è multi-paradigma, ma fortemente orientato agli oggetti: perfino i tipi primitivi e i metodi vengono trattati come oggetti. Aggiungendo la possibilità di estendere ogni classe con nuovi metodi, anche successivamente alla sua dichiarazione, si ottiene un linguaggio estremamente flessibile ed estendibile, che consente di adattare il comportamento del linguaggio stesso alle proprie necessità.

Da queste premesse si intuisce come Ruby sia predisposto per essere facilmente estensibile, infatti è presente un RubyGems, il *package manager* per gestire la distribuzione delle *gemme*, pacchetti che possono includere varie funzionalità per estendere il linguaggio, tra cui librerie o eseguibili. La configurazione delle gemme necessarie per un progetto, con le loro dipendenze e versioni, viene eseguita da Bundler, che si occupa di mantenere tutte queste informazioni nei *Gemfile*. Di seguito sono elencate le principali gemme utilizzate nel progetto di stage.

Rails Framework utilizzato, illustrato successivamente in dettaglio.

Devise Fornisce molti metodi e endpoint già configurati per l'autenticazione sicura degli utenti, insieme alla gemma Devise Token Auth, che fornisce l'autenticazione tramite token.

Pundit Permette di separare la gestione dei permessi degli utenti dai controller, riducendo anche il codice ripetuto per l'autorizzazione.

2.4.2 Rails

Alcune gemme di RubyGems sono molto complete, tanto da includere all'interno un intero framework. L'esempio principale è la gemma Rails, che fornisce il framework *Ruby on Rails*[2], o *Rails*. Si basa sul paradigma *convention over configuration*, quindi cerca di ridurre al minimo le decisioni che lo sviluppatore deve prendere, al fine di ridurre il codice *boilerplate* e favorire il rispetto del principio *don't repeat yourself* (DRY).

Rails fornisce tutti gli strumenti per sviluppare un'applicazione web basata sul pattern Model-View-Controller (MVC), in particolare le librerie:

Active Record Implementa l'omonimo pattern architetturale, in cui si utilizza un database relazionale per memorizzare le informazioni relative ai modelli. Fornisce un'interfaccia per astrarre svariati *database management system* (DBMS), in particolare una contenuta nella gemma PG, che permette di utilizzare un database basato su PostgreSQL, comunemente utilizzato nei progetti di Moku.

Action Controller Fornisce metodi e classi per la definizione dei controller REST dell'applicazione.

Action View Fornisce metodi e classi per la definizione della parte di view dell'applicazione. Nei progetti di Moku non viene generalmente utilizzata, perché si utilizza Rails per sviluppare API backend, che comunicano con view separate. Nel caso del progetto di stage il frontend è sviluppato con Angular.

[2] *Sito ufficiale di Ruby on Rails.* URL: https://rubyonrails.org/.

Active Storage Permette di memorizzare file, come immagini o documenti, su servizi cloud, invece che sul server o nel database. Sono disponibili diversi servizi, Moku utilizza AWS S3.

Active Admin Permette di creare velocemente e con semplicità interfacce per la gestione dei dati nel database. Non è stato usato nel progetto di stage, perché il frontend era già progettato per fornire le stesse funzionalità agli utenti super admin.

2.4.3 API REST

REST è uno stile architetturale per la progettazione di API *stateless*. Si basa su chiamate a degli *endpoint* (*routes* in Rails), coppie formate da un URL, che identifica univocamente una risorsa, e un metodo HTTP, normalmente uno tra GET, POST, PUT, PATCH o DELETE. Viene ampiamente utilizzato nello sviluppo di API ed è supportato nativamente da molti framework.

Essendo uno stile architetturale, REST non è una vera e propria tecnologia, ma in questo caso si contrappone a GraphQL, un linguaggio di query per API, che viene normalmente utilizzato nei progetti di Moku. Per il progetto di stage si è optato per sviluppare una API REST per limitare le modifiche necessarie da fare sul frontend, essendo che questo era già pronto e si basava sulla API precedente, sviluppata secondo lo stile REST. Sarebbe stato troppo costoso modificarne il funzionamento perché potesse funzionare con GraphQL.

Capitolo 3

Analisi e refactor dei modelli

3.1 Introduzione

La prima attività che ho dovuto svolgere è stata la definizione dei modelli del nuovo backend. Purtroppo l'architettura già presente nel vecchio backend, soprattutto lato database, aveva diversi difetti e andava rivista interamente, quindi è stato necessario partire da un'analisi di quanto già fatto, per capire se fosse necessario effettuare del refactor nella struttura dei dati. In molti casi le entità contenevano molti attributi superflui o non adatti ad essere associati alla specifica entità, perché più adatti ad essere associati attraverso una relazione a un'altra entità. Le decisioni sono state prese anche basandosi sull'analisi dei requisiti già fatta precedentemente al mio arrivo. Questa è stata tradotta in *user stories*, che hanno guidato la successiva fase di codifica.

L'architettura per il nuovo backend è quella inferita dalle convenzioni di Rails, soprattutto a causa dell'applicazione del pattern Active Record. Essendo le classi dei modelli dell'applicazione strettamente legate alle entità presenti nel database, la loro progettazione è stata fatta proseguire parallelamente, al punto che in molti casi è possibile riferirsi interscambiabilmente a attributi di un'entità o del modello associato. Per riferirsi a modelli ed entità verranno seguite le convenzioni di Rails: i nomi dei modelli sono scritti al singolare in `PascalCase`, mentre quelli delle entità sono al plurale in `snake_case`, ad esempio al modello `EventParticipation` è associata l'entità `event_participations`.

In questo capitolo vengono riportati i cambiamenti e le scelte più significative che sono state fatte durante questa fase, che hanno portato alla produzione del diagramma ER completo dell'applicazione, riportato nell'appendice A.

3.2 Modifiche significative effettuate

3.2.1 Utenti

La prima tabella analizzata è stata quella degli utenti. Nella versione precedente le informazioni relative al modello degli utenti erano divise in tre entità:

- `e_users`: conteneva i dettagli principali degli utenti, come nome, cognome e email, i campi per la gestione dell'autenticazione e altri, che però erano inutilizzati;
- `e_partecipants`: era dedicata agli utenti partecipanti. Conteneva un sottoinsieme dei campi di `e_users`:

    - `companyId`,
    - `firstName`,
    - `lastName`,
    - `email`,
    - `phone`,
    - `createdAt`,
    - `status`.

- `e_users_levels`: memorizzava i ruoli degli utenti e, per ogni azione esposta dall'API, un campo booleano che indicava se quel ruolo avesse l'autorizzazione di effettuare quell'azione.

Ho deciso di raccogliere queste informazioni in un unico modello `User`, con i campi dell'entità `e_partecipants`, a cui ho aggiunto l'immagine dell'utente e un enumerazione che ne indica il ruolo, invece di utilizzare un'intera entità come era stato fatto precedentemente. I campi necessari per l'autenticazione sono stati omessi, perché già presenti nel *template* dei progetti di Moku, che è stato usato come base di partenza anche per questo.

La motivazione alla base della scelta di non codificare le autorizzazioni all'interno di un modello è stata la complessità del processo di autorizzazione nel dominio trattato, che spesso dipende anche dall'oggetto su cui si vuole eseguire un'azione, qundi non esclusivamente dal ruolo dell'utente.

La scelta di unire i partecipanti (corretti in "participants", invece di "partecipants") al resto degli utenti è stata presa per ridurre la duplicazione dei dati, in questo caso rilevante, perché le due entità, una volta riviste, avrebbero avuto quasi tutti i campi in comune. Inoltre, seppur vero che i partecipanti non possono effettuare l'autenticazione nell'applicazione web, utilizzano l'applicazione *mobile*, che utilizzerà lo stesso backend.

3.2.2 Integrazioni

L'aspetto che più è stato cambiato nella nuova versione è il modo in cui vengono memorizzate le informazioni relative alle integrazioni. Nella versione precedente queste non erano strutturate in alcun modo: erano contenute in ben 4 entità diverse, pensate per essere dedicate ad altri tipi di dati, come intuibile dai nomi:

- `e_platforms`: qui venivano salvate le integrazioni abilitate per la piattaforma, cioè disponibili a tutti gli organizzatori sotto quella specifica piattaforma. L'informazione era memorizzata attraverso in un attributo booleano per ogni integrazione;

- `e_companies`: qui erano contenuti i token di accesso alle API delle integrazioni ed eventuali chiavi o link necessari;

- `e_events`: qui venivano memorizzati id, chiavi o altri dettagli dei meeting dell'evento;

- `e_events_users_link`: qui venivano salvati alcuni link per l'accesso al meeting dell'evento.

Come si può vedere, l'informazione era distribuita su diversi attributi, che avrebbero potuto essere rifattorizzati in entità separate, così da poter permettere anche l'aggiunta, la rimozione o la modifica delle varie integrazioni disponibili, invece di mantenerle *hardcoded* nel database. Questo ha portato alla creazione di 3 entità interamente dedicate alle integrazioni:

- `integrations`: contiene i nomi delle integrazioni disponibili nell'applicazione. Le piattaforme dichiarano le integrazioni utilizzate basandosi su quelle presenti in questa entità;
- `organizer_integrations`: memorizza le integrazioni rese disponibili per ogni organizzatore, tra quelle disponibili per la sua piattaforma, e i dettagli dell'integrazione, necessari agli organizzatori;
- `event_integrations`: memorizza le integrazioni rese disponibili per ogni evento, tra quelle disponibili per il suo organizzatore, e i dettagli dell'integrazione necessari all'evento;
- `active_campaign_event_integration_details`: contiene i dettagli aggiuntivi, necessari all'integrazione ActiveCampaign.

Le entità create non sono definitive, soprattutto per quanto riguarda gli attributi. Una volta effettuata un'analisi più approfondita di ogni integrazione e sullo scopo di ognuna, si valuterà se rivedere questa struttura.

3.2.3 Piattaforme

Oltre agli attributi relativi alle integrazioni, sono stati rimossi dall'entità `platforms` anche quelli che permettevano la configurazione personalizzata di un server SMTP per ogni piattaforma. La scelta è stata motivata dalla presenza di alcuni casi, già accaduti, in cui questa configurazione non era stata fatta correttamente, portando a notifiche mail non funzionanti. Il motivo dell'accaduto era stata la poca formazione specifica della persona che aveva configurato la piattaforma. Non potendo fare assunzioni sulla persona che avrebbe configurato le piattaforme si è preferito centralizzare il procedimento in Moku: i clienti che desiderano personalizzare le impostazioni SMTP per la loro piattaforma potranno chiedere all'azienda di configurarle, secondo le loro richieste. In questo modo si può garantire il funzionamento delle notifiche mail, essenziali anche solamente per registrare un nuovo utente.

3.2.4 Eventi

L'entità `e_events`, associata al modello degli eventi, era la più estesa del sistema, com'è lecito aspettarsi, essendo questa la funzionalità fulcro dell'intera applicazione. Non è però giustificabile la complessità e la quantità di informazioni che venivano memorizzate nell'entità: molte riguardavano le integrazioni, già rifattorizzate in entità separate, altre saltavano subito all'occhio per essere replicate tre volte:

- `reminderNDate`,
- `reminderNSent`,
- `reminderNEmailText`,
- `idTemplateEmailReminderN`,

- `reminderNEmailTextOnline`,
- `idTemplateEmailReminderNOnline`.

Sostituendo $n \in \{1, 2, 3\}$ a N, si ottengono tutti gli attributi replicati, ad esempio `reminderNSent` era presente come `reminder1Sent`, `reminder2Sent` e `reminder3Sent`. Tutto questo portava l'entità ad avere un totale di ben 84 attributi, un numero decisamente troppo elevato per essere gestibile nel modello di una funzionalità simile. Gli attributi replicati sono stati rifattorizzati in una nuova entità `event_reminders`:

- `date`,
- `sent`,
- `target`,
- `email_template_id`,
- `email_message`,
- `event_id`.

La creazione di `event_reminders` è stata motivata dalla possibilità di rendere flessibile il numero di *reminder*, invece di mantenerne uno fisso arbitrario, all'interno di un'entità non strettamente dedicata a rappresentare quei dati. Oltretutto l'entità separata permette di avere un numero variabile di *reminder* tra gli eventi, evitando ridondanza. Il frontend permetteva già di scegliere il numero di *reminder* da inviare, ma se questo era minore di 3, gli attributi duplicati venivano semplicemente resi `NULL`. Infine, la nuova entità prevede l'attributo `target`, che specifica se il messaggio è per gli utenti che parteciperanno fisicamente o online all'evento, evitando ridondanza nel caso in cui si voglia impostare il *reminder* solo per gli uni, piuttosto che per gli altri.

Nonostante queste modifiche il modello `Event` rimane abbastanza complesso, quindi non si escludono modifiche future, una volta arrivati alla sua implementazione. È probabile che, con l'avanzamento della codifica, si noti la necessità di ulteriori modifiche e ottimizzazioni, anche grazie a una conoscenza più approfondita del dominio. Per la durata del mio stage questo non è stato possibile, quindi viene riportata nel diagramma ER in appendice la mia proposta pensata nella fase di progettazione, che non va intesa come una versione definitiva dell'entità.

3.2.5 Partecipazioni

Nella vecchia versione del backend i partecipanti non venivano considerati dei normali utenti del sistema, ma venivano salvati separatamente, nell'entità `e_partecipants`. Come discusso in §3.2.1, ora anche i partecipanti vengono gestiti nel modello `User`, per cui le partecipazioni agli eventi dovranno avere una relazione con l'entità `users`. Questo permette anche alle altre categorie di utenti di partecipare a un evento, senza creare un account separato con il ruolo di partecipante.

Rispetto all'entità originale `e_events_users_link`, che si occupava di rappresentare le partecipazioni a uno specifico evento, sono stati rimossi i link di accesso ai meeting, spostati nelle entità delle integrazioni, come descritto in §3.2.2.

Un ultimo dettaglio modificato, rispetto all'entità originale, è la relazione con l'entità `event_participation_credits`. Questa si occupa di tenere traccia dei crediti guadagnati da un utente nella partecipazione a un evento. Prima questa si riferiva

direttamente all'evento interessato, ma essendo i crediti dipendenti dalla partecipazione dell'utente all'evento, la modifica è sembrata logica. In alcuni casi i crediti "guadagnati" possono essere negativi, ad esempio nel caso in cui un utente non partecipi effettivamente all'evento, quindi senza eseguire il check-in. In questo caso il sistema funziona lo stesso, perché il modello `EventParticipation` tiene solamente traccia delle iscrizioni a un evento, quindi alla volontà da parte dell'utente di partecipare; il tracciamento dell'effettiva partecipazione e della sua durata viene delegato al modello `EventParticipationOperation`.

3.2.6 Altre modifiche minori

Sono state effettuate anche altre modifiche minori nei modelli che non sono stati citati. Di queste non ritengo sia utile una descrizione dettagliata, vengono solamente riportate di seguito:

- assegnati nomi più intuitivi e coerenti con il dominio ad attributi e modelli, ad esempio gli organizzatori venivano chiamati *companies*. Questo creava confusione, perché il nome non corrispondeva al termine utilizzato nel frontend;
- rimossi gli attributi inutilizzati;
- alcuni attributi booleani sono stati resi uno stato del modello, ad esempio `e_message` possedeva l'attributo `isLetto` per segnalare la lettura di un messaggio, mentre la nuova entità `conversation_messages` lo segnala impostando il valore `read` all'enumerazione `status`;
- assegnati tipi più espliciti agli attributi: molte entità possedevano diverse stringhe, quando queste potevano essere espresse con enumerazioni, date, testi o numeri;
- rimossi i *counter*: alcuni modelli avevano la necessità di tenere traccia dell'ordine di creazione dei record, quindi ricorrevano ad attributi ad auto-incremento, perché gli identificativi erano testuali. In Rails i modelli possiedono sempre un identificativo ad auto-incremento, quindi l'attributo aggiuntivo risulta superfluo.

Capitolo 4

Progettazione della API

4.1 Introduzione

L'API è la parte più importante del backend: è l'interfaccia che viene esposta per interagire con l'applicazione. Viene utilizzata dal frontend per ottenere i dati necessari a soddisfare le richieste degli utenti, che, essendo già sviluppato, si aspetta di poter eseguire certe richieste e di ricevere una risposta di un certo tipo. È previsto che il frontend debba adattarsi ai cambiamenti del nuovo backend, una volta pronto, ma questi dovrebbero essere ridotti al minimo, quindi nella progettazione dei controller ho cercato di trovare il giusto equilibrio con quanto era stato fatto in precedenza e lo stile che viene suggerito dal framework.

4.2 Notazione adottata

Prima di descrivere la struttura delle operazioni viene illustrata la notazione adottata per gli endpoint:

- la prima parte indica il metodo HTTP utilizzato: uno tra GET, POST, PUT, PATCH e DELETE, negli endpoint realizzati;
- la seconda parte indica l'URI della risorsa relativa all'endpoint, senza specificare l'URL dell'applicazione;
- tutti gli URI degli endpoint implementati sono all'interno del percorso `/api`;
- alcuni endpoint si riferiscono a una specifica risorsa invece che a una lista o collezione. In tal caso l'identificativo della risorsa viene rappresentato con la notazione `:id` o `:resource_id`;
- viene utilizzata la parola `resources` per rappresentare il nome di un modello generico.

Ad esempio:

1. una chiamata con metodo HTTP PUT a `https://app.evvvents.it/api/platforms/2` risponde all'endpoint `PUT /api/platforms/:id`;
2. a un endpoint generico `DELETE /api/resources/:id` può rispondere alle chiamate `DELETE /api/locations/1` o `DELETE /api/users/4`.

Ad ogni risposta viene associato uno stato HTTP. In caso di successo viene sempre restituito il codice `200`, mentre gli errori restituiscono codici diversi, in base al contesto. I codici di errore generali, che possono essere restituiti da qualsiasi endpoint implementato sono:

- `404`, nel caso in cui l'endpoint o la risorsa richiesta non esista;
- `500`, nel caso di errori generici nel backend, che non sono stati gestiti esplicitamente.

Gli altri casi di errore vengono descritti nelle sezioni successive.

4.3 Descrizione delle funzionalità esposte

Di seguito vengono descritte le operazioni che deve essere possibile eseguire su ogni modello dell'applicazione, chiamando i relativi endpoint.

4.3.1 Lista delle risorse

Risponde all'endpoint `GET /api/resources`, restituendo la lista delle risorse associate al modello specificato, generalmente ordinate per nome, dove presente. La risposta prevede l'utilizzo della paginazione, con cui si può interagire attraverso i due parametri GET opzionali:

1. `page`, che indica il numero della pagina che si vuole visualizzare. Se non indicato, la risposta mostrerà la prima pagina;
2. `per_page`, che indica il numero di risorse presenti per pagina. Se non indicato vengono mostrate 25 risorse per pagina, valore già utilizzato nel frontend.

Di ogni risorsa vengono mostrate solo le informazioni essenziali, invece di tutte quelle fornite dal modello. Di seguito vengono elencati gli attributi restituiti per ogni risorsa, nella lista di ogni modello implementato:

User

- `id`,
- `email`,
- `first_name`,
- `last_name`,
- `role`,
- `organizer_id`

Plan

- `id`,
- `name`,
- `description`,
- `price`

Platform

- `id`,
- `name`,
- `website`,
- `host_url`,
- `main_organizer_id`

Organizer

- `id`,
- `name`,
- `platform_id`,
- `address`,
- `city`,
- `country`

Location

- `id`,
- `name`,
- `organizer_id`

4.3.2 Dettagli di una risorsa

Risponde all'endpoint `GET /api/resources/:id`, restituendo i dettagli della risorsa specificata. I dettagli non comprendono la totalità degli attributi del modello, ma un suo sottoinsieme selezionato, per nascondere all'utente eventuali dettagli implementativi non rilevanti o attributi di tracciamento, come il creatore o l'istante di creazione e ultima modifica del record. Di seguito vengono elencati, per ogni modello, gli attributi restituiti per ogni risorsa:

User

- `id`,
- `email`,
- `image`,
- `first_name`,
- `last_name`,
- `role`,
- `phone`,

- status,
- organizer_id

Plan

- id,
- name,
- price,
- description,
- max_events,
- max_participants,
- max_integrations,
- max_storage,
- desk_number

Platform

- id,
- name,
- logo,
- logo_dark,
- favicon,
- logo_small,
- website,
- host_url,
- privacy_url,
- powered_by,
- action_bar_color,
- side_bar_color,
- border_color,
- sign_in_color,
- event_color,
- status,
- plan_id,
- main_organizer_id

Organizer

- `id`,
- `name`,
- `logo`,
- `platform_id`,
- `vat`,
- `sdi`,
- `address`,
- `city`,
- `country`,
- `zip_code`,
- `status`,
- `email`,
- `email_pec`,
- `phone`,
- `finished_webhook`,
- `add_participant_webhook`,
- `remove_participant_webhook`

Location

- `id`,
- `name`,
- `status`,
- `description`,
- `image`,
- `address`,
- `civic`,
- `city`,
- `province`,
- `zip_code`,
- `google_place_identifier`,

- `latitude`,
- `longitude`,
- `how_to_get_by_car`,
- `how_to_get_by_plane`,
- `how_to_get_by_train`,
- `internal_notes`,
- `organizer_id`

4.3.3 Creazione di una risorsa

Risponde all'endpoint `POST /api/resources`, creando un nuovo record del relativo modello, con gli attributi specificati nel corpo della richiesta. Restituisce nella risposta i dettagli della risorsa appena creata, se l'operazione è avvenuta con successo.

4.3.4 Modifica di una risorsa

Risponde agli endpoint `PUT /api/resources/:id` e `PATCH /api/resources/:id`, modificando gli attributi della risorsa, specificati nel corpo della richiesta, con i valori passati. Restituisce nella risposta i dettagli della risorsa appena modificata, se l'operazione è avvenuta con successo.

4.3.5 Eliminazione di una risorsa

Risponde all'endpoint `DELETE /api/resources/:id`, eliminando la risorsa fisicamente dal database oppure logicamente, impostando lo stato del record ad `archived`, nei modelli in cui è prevista questa modalità di cancellazione. Restituisce nella risposta i dettagli della risorsa appena eliminata, se l'operazione è avvenuta con successo.

4.3.6 Lista dei ruoli degli utenti

Risponde all'endpoint `GET /api/users/roles`, elencando tutti i possibili ruoli di un utente:

- super admin;
- admin;
- manager;
- staff;
- speaker;
- participant.

Questa è un'informazione utile al frontend, che utilizza già estensivamente, quindi andava fornita attraverso un endpoint dedicato, non potendo essere reperita in altri modi. Per la struttura attuale dell'applicazione la risposta è sempre la stessa, perché i ruoli sono *hardcoded* nel software.

4.3.7 Lista delle risorse interne a una specifica risorsa

Alcuni modelli sono logicamente contenuti all'interno di altri, si vuole quindi poter elencare tutte le risorse contenute in una seconda risorsa. Nei modelli che sono stati implementati durante il progetto di stage questo accade in due casi:

1. gli organizzatori stanno all'interno di una specifica piattaforma;
2. ogni organizzatore crea e utilizza delle *location*.

In questi casi se si vogliono elencare le risorse `rescources_b` contenute nella risorsa `resource_a` con identificativo `:id` si effettua una chiamata all'endpoint `GET /api/resources_a/:id/resources_b`, ad esempio per elencare tutti gli organizzatori della piattaforma con id 10 si chiama `GET /api/platforms/10/organizers`. È comunque possibile ottenere la lista di tutte le risorse relative a un modello, anche quando questo stia logicamente all'interno di un'altra risorsa. Ad esempio è comunque possibile ottenere la lista di tutti gli organizzatori di tutte le piattaforme chiamando `GET /api/organizers`.

4.3.8 Creazione di una risorsa all'interno di un'altra risorsa

Nei casi appena sopracitati è necessario specificare la risorsa in cui sarà contenuta quella che si intende creare, quindi ho ritenuto opportuno rendere necessaria la creazione di queste particolari risorse chiamando l'endpoint `POST /api/resources_a/:id/resources_b`, invece che con l'endpoint descritto nella sotto-sezione 4.3.3, che non viene esposto per i modelli interessati.

4.4 Gestione dei permessi

L'API, oltre ad essere utilizzata dal frontend, potrebbe essere usata da qualsiasi altro tipo di *client*, con o senza buone intenzioni, quindi è fondamentale che siano presenti dei controlli sulle autorizzazioni degli utenti che effettuano richieste, per ogni funzionalità esposta.

4.4.1 Autenticazione e autorizzazione

Prima di procedere bisogna fare un'importante premessa: autenticazione e autorizzazione, seppur simili, sono due concetti diversi e, come tali, devono essere separati anche strutturalmente.

L'autenticazione avviene quando l'utente esegue l'azione tecnicamente chiamata *sign-in* o *log-in*. Serve a provare che l'utente è chi dichiara di essere. A livello di codifica questa viene gestita da Devise, una gemma adatta allo scopo, dunque non è stato necessario definirne la progettazione.

L'autorizzazione definisce se uno specifico utente o tipologia di utenti abbia o meno il permesso di eseguire una certa azione. Questa sezione si occupa di descrivere le autorizzazioni richieste per ogni funzionalità implementata dall'API, riferita nel modo seguente:

- **lista**: si riferisce a quanto descritto in §4.3.1;
- **lista nella risorsa**: si riferisce a quanto descritto in §4.3.7;
- **dettaglio**: si riferisce a quanto descritto in §4.3.2;

- **creazione**: si riferisce a quanto descritto in §4.3.3 e in §4.3.8;
- **modifica**: si riferisce a quanto descritto in §4.3.4;
- **eliminazione**: si riferisce a quanto descritto in §4.3.5.

Nella maggior parte dei casi non sarà necessario entrare così nel dettaglio, perché spesso le azioni di lista e di dettaglio sono accomunate nella categoria delle azioni di lettura, mentre quelle di creazione, modifica ed eliminazione vengono categorizzate come azioni di scrittura, tutte con gli stessi permessi di base, a cui eventualmente si aggiungono ulteriori vincoli o permessi.

4.4.2 Permessi richiesti

I permessi richiesti dalle azioni sui modelli vengono definite in base al ruolo dell'utente autenticato nel sistema al momento della richiesta. La prima condizione per qualsiasi azione quindi, è che l'utente abbia eseguito con successo l'autenticazione (azione per cui ovviamente la condizione non vale). L'azione stessa di autenticazione richiede delle autorizzazioni: sono tutte gestite da Devise, tranne la condizione che prevede che un utente non possa eseguire il *log-in* se ha il ruolo di partecipante, perché l'applicazione web è dedicata solamente alla gestione degli eventi, a cui i partecipanti non devono avere accesso: loro si interfacciano al sistema esclusivamente attraverso l'applicazione *mobile*.

User

L'autorizzazione delle azioni eseguite da un utente su un altro utente sono particolarmente complicate, perché dipendono dalla combinazione dei ruoli dell'utente interessato e dell'utente che ha eseguito la richiesta.

- **lettura**: tutti gli utenti possono eseguire questa azione, ognuno però vede solamente gli utenti diversi da se stesso, con ruolo uguale o inferiore all'utente che ha eseguito la richiesta. A questo si aggiunge che:
  - l'admin vede solamente utenti della sua stessa piattaforma;
  - il manager e lo staff vedono solamente utenti della loro stessa organizzatore;
  - lo speaker vede solo partecipanti e non può essere visto da nessuno, perché dipende dal modello degli eventi, che non è stato implementato durante lo svolgimento del progetto di stage.
- **dettaglio**: ai permessi di lettura si aggiunge il permesso di un utente di vedere i dettagli di se stesso;
- **scrittura**: i permessi sono gli stessi della lettura, ma in questo caso il ruolo dell'utente che esegue la richiesta deve essere superiore in senso stretto;
- **modifica**: ai permessi di scrittura si aggiunge il permesso, per un utente, di modificare se stesso;
- **eliminazione**: ai permessi di scrittura si aggiunge il permesso, per un utente, di eliminare se stesso, nel solo caso in cui questo sia uno speaker;
- **lista dei ruoli**: si riferisce a §4.3.6. Non è richiesta né l'autorizzazione, né l'autenticazione.

Plan

- **lettura**: concessa a tutti gli utenti;
- **scrittura**: permessa solamente ai super admin.

Platform

- **lista**: permessa solamente ai super admin;
- **dettaglio**: permesso ai super admin e agli admin della piattaforma su cui viene eseguita la richiesta;
- **scrittura**: permessa solamente ai super admin.

Organizer

- **lista**: permessa solamente ai super admin;
- **lista nella piattaforma**: permessa ai super admin e agli admin della piattaforma;
- **dettaglio**: come la lista nella piattaforma, ma è permesso anche al manager dell'organizzatore;
- **scrittura**: come la lista nella piattaforma.

Location

- **lista**: permessa solamente ai super admin;
- **lista nell'organizzatore**: come il dettaglio per il modello **Organizer**;
- **dettaglio**: come la lista nell'organizzatore;
- **scrittura**: come la lista nell'organizzatore;

4.4.3 Parametri permessi

L'autorizzazione deve anche occuparsi dei parametri che possono essere permessi nel corpo delle richieste di creazione e modifica delle risorse. Vengono elencati di seguito per ogni modello:

User

Per la creazione:

- `email`,
- `first_name`,
- `last_name`,
- `role`,
- `phone`,
- `password`,

- `organizer_id`,
- `attachments`:
    - `image`

Per la modifica sono permessi gli stessi parametri consentiti per la creazione, eccetto la password e, se un utente sta modificando se stesso, il ruolo.

Plan

- `name`,
- `description`,
- `max_events`,
- `max_participants`,
- `max_integrations`,
- `max_storage`,
- `desk_number`,
- `price`

Platform

- `name`,
- `website`,
- `host_url`,
- `privacy_url`,
- `powered_by`,
- `action_bar_color`,
- `side_bar_color`,
- `border_color`,
- `sign_in_color`,
- `event_color`,
- `main_organizer`,
- `plan_id`,
- `attachments`:
    - `logo`,
    - `logo_dark`,
    - `favicon`,
    - `logo_small`

Organizer

- name,
- vat,
- sdi,
- address,
- city,
- country,
- zip_code,
- email,
- email_pec,
- phone,
- finished_webhook,
- add_participant_webhook,
- remove_participant_webhook,
- attachments:
  - logo

Location

- name,
- description,
- address,
- civic,
- city,
- province,
- zip_code,
- google_place_identifier,
- latitude,
- longitude,
- how_to_get_by_car,
- how_to_get_by_plane,
- how_to_get_by_train,
- internal_notes,
- attachments:
  - image

Capitolo 5

Codifica

5.1 Modelli

La codifica dei modelli passa per tre fasi successive:

1. creazione della entità del modello nel database e della classe, utilizzando le migrazioni;
2. l'associazione del modello con altri modelli o elementi di *storage*;
3. le validazioni sugli attributi e sulle associazioni dichiarate.

5.1.1 Migrazioni del database

Basandosi su quanto definito nella fase di progettazione dei modelli, descritta nel capitolo 3, questi sono stati generati utilizzando da linea di comando il generatore automatico `rails generate model` o, in versione ridotta, `rails g model`. Il comando accetta come argomenti:

- il nome del modello, al singolare e in *CamelCase*;
- gli attributi che deve avere il modello;
- per ogni attributo: il suo tipo, che rispecchia, ad alto livello, i tipi comunemente disponibili per le colonne nei DBMS SQL. Normalmente è uno dei seguenti tipi nativi delle migrazioni di Rails[1], agnostici rispetto all'implementazione del database:
  - `primary_key`,
  - `string`,
  - `text`,
  - `integer`,
  - `bigint`,
  - `float`,
  - `decimal`,

[1] *Documentazione ufficiale di Ruby on Rails - metodo* ***`add_column`***. URL: https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html#method-i-add_column.

  - `datetime`,
  - `timestamp`,
  - `time`,
  - `date`,
  - `binary`,
  - `blob`,
  - `boolean`,
  - `references`

- per ogni attributo: l'identificatore `uniq`, che imposta un indice su quella colonna del database, che ne specifica l'unicità nell'entità.

Di conseguenza, la sintassi generale è la seguente:

```
rails g model ModelName attr_1:type:[uniq] attr_2:type:[uniq] ...
```

Codice 3: Comando `rails generate model`

Portando un esempio, per la generazione di una versione semplificata del modello degli organizzatori può essere usato il comando seguente:

```
rails g model Organizer name:string vat:string:uniq address:string
↪  platform:references creator_id:bigint
```

Codice 4: Esempio di utilizzo del comando `rails generate model`

che genera la seguente migrazione:

```ruby
# db/migrate/{timestamp}_create_organizers.rb

class CreateOrganizers < ActiveRecord::Migration[7.0]
  def change
  create_table :organizers do |t|
    t.string :name
    t.string :vat
    t.string :address
    t.references :platform, foreign_key: true
    t.bigint :creator_id

    t.timestamps
  end
  add_index :organizers, :vat, unique: true
  end
end
```

Codice 5: Migrazione generata dal comando `rails generate model`

Successivamente questa viene modificata, aggiungendo i vincoli `NOT NULL` e la chiave esterna verso l'utente creatore prima di eseguire effettivamente la migrazione.

Si noti come non sia necessario specificare la chiave primaria. Il comportamento di *default* di Active Record è introdurre automaticamente un identificativo progressivo, chiamato `id`, di tipo `bigint`. Inoltre `timestamps` genera automaticamente degli attributi gestiti dalla gemma, per tracciare l'istante di creazione e ultima modifica dei record.

La migrazione dopo la modifica è la seguente:

```ruby
# db/migrate/{timestamp}_create_organizers.rb

class CreateOrganizers < ActiveRecord::Migration[7.0]
  def change
    create_table :organizers do |t|
      t.string :name, null: false
      t.string :vat, null: false
      t.string :address, null: false
      t.references :platform, null: false, foreign_key: true
      t.bigint :creator_id, null: false

      t.timestamps
    end
    add_foreign_key :organizers, :users, column: :creator_id
    add_index :organizers, :vat, unique: true
  end
end
```

Codice 6: Migrazione modificata manualmente

Utilizzando il metodo `change`, le migrazioni possono modificare la struttura del database secondo quando specificato nella migrazione, senza necessità di ricorrere a *downtime* del server e di eseguire il rollback alla versione dello schema del database precedente, se fosse necessario.

Il generatore produce altri due file, oltre alla migrazione:

```ruby
# app/models/organizer.rb

class Organizer < ActiveRecord::Base
  belongs_to :platform
end
```

Codice 7: File del modello generato dal comando `rails generate model`

```ruby
# spec/models/organizer_spec.rb

require 'rails_helper'
RSpec.describe Organizer, type: :model do
```

```
  pending "add some examples to (or delete) #{__FILE__}"
end
```

Codice 8: File di spec generato dal comando `rails generate model`

Il primo contiene la definizione della classe, in cui andranno inseriti i metodi, le validazioni e le associazioni sul modello, descritte in §5.1.2 e §5.1.3. Nel secondo andranno definiti i test di unità per il modello, descritti in §5.4.

5.1.2 Associazioni a modelli e file

Una volta generata la struttura del modello attraverso le migrazioni del database, è stato necessario associare tra loro i modelli, al livello dell'applicazione, secondo le relazioni espresse nel diagramma ER prodotto durante la fase di analisi e refactor (§3). Per farlo, sono stati utilizzati i metodi forniti da `ActiveRecord::Base`, classe ereditata da tutti i modelli. Nel progetto, in realtà, tutti i modelli ereditano dalla classe `ApplicationRecord`, che a sua volta eredita da `ActiveRecord::Base`, ma viene utilizzato per aggiunge metodi di utilità comuni a tutti i modelli implementati.

Rails incentiva l'implementazione di associazioni bidirezionali, attraverso l'utilizzo dei metodi:

- `belongs_to`: utilizzato per specificare l'associazione con il modello di cui la classe memorizza la chiave esterna;
- `has_one`: specifica l'associazione con un record di un modello che memorizza la chiave esterna alla classe;
- `has_many`: specifica l'associazione con più record di un modello che memorizza la chiave esterna alla classe;
- `has_and_belongs_to_many`: permette di specificare associazioni del tipo "molti a molti", utilizzando una tabella, creata manualmente, che possiede le chiavi esterne ad entrambi i modelli coinvolti.

Oltre alle associazioni con i modelli sono state specificate le associazioni con i file. Queste vengono gestite con la gemma Active Storage, che fornisce i metodi per eseguire l'associazione (*attach*) dei file, chiamati *attachments*: `has_one_attached` e `has_many_attached`.

Proseguendo con l'esempio dell'implementazione degli organizzatori, il file della classe `Organizer` con le associazioni dichiarate risulta essere il seguente:

```
# app/models/organizer.rb

class Organizer < ApplicationRecord
  belongs_to :platform
  belongs_to :creator, class_name: 'User', inverse_of: :created_organizers,
  ↪  optional: true
```

```
  has_many :locations, dependent: :nullify

  has_one_attached :logo
end
```

Codice 9: Classe `Organizer` con le associazioni

L'associazione `has_many` è l'altra estremità di un'associazione `belongs_to` dichiarata nel modello `Location` e si basa sulla convenzione dei nomi di Rails per cui `Location` deve avere una colonna nel database chiamata `organizer_id`, che punta all'identificativo di un organizzatore. Serve per rendere accessibili le *location* gestite dall'organizzatore, avendo il record dell'organizzatore stesso.

5.1.3 Validazioni

L'ultima fase di sviluppo della parte di modello sono le validazioni sugli attributi e le associazioni, che vengono codificate utilizzando principalmente i validatori forniti da Active Record, da altre gemme o creati manualmente. Le validazioni inseriscono dei messaggi di errore in un array associato al modello che si sta validando; prima di memorizzare nel database i dati del record che si vuole salvare, ad esempio chiamando i metodi `save` o `create`, viene chiamato il metodo `valid?`, che controlla che l'array di errori sia vuoto, altrimenti è possibile accedervi e mostrare gli errori, anche come risposta della API. Le validazioni fornite da Active Record permettono di verificare vari aspetti degli attributi, come la presenza di un valore (`NOT NULL`), il formato di una stinga o i controlli sui valori numerici e molti altri.

Aggiungendo le validazioni sugli attributi, il file di esempio degli organizzatori diventa:

```
# app/models/organizer.rb

class Organizer < ApplicationRecord
belongs_to :platform
  belongs_to :creator, class_name: 'User', inverse_of: :created_organizers,
  ↪  optional: true

  has_many :locations, dependent: :nullify
  has_one_attached :logo
  validates :logo, attached: true, content_type: ['image/png', 'image/jpeg']

  validates :name,
            :vat,
            :address,
            presence: true
```

```
  validates :vat, uniqueness: true

  validates :creator, presence: true, on: :create
  validate :creator_not_changed

private

  def creator_not_changed
    errors.add :creator, :cannot_change if creator_changed? && persisted?
  end
end
```

Codice 10: Classe `Organizer` con le validazioni

5.1.4 Associazione a `creator`

Quasi tutti i modelli implementati nel corso del progetto hanno un'associazione con il modello degli utenti, che traccia l'utente che ha creato il record. Sono necessarie diverse righe di codice per implementare questa funzionalità, non sempre intuitive oltretutto, quindi ho deciso di incapsulare questa configurazione all'interno di un concern, una funzionalità offerta da Active Support, che sfrutta i moduli di Ruby. Dopo l'operazione di refactor il file del modello di esempio diventa:

```
# app/models/organizer.rb

class Organizer < ApplicationRecord
  include Creator
  belongs_to :platform

  has_many :locations, dependent: :nullify

  has_one_attached :logo
  validates :logo, attached: true, content_type: ['image/png', 'image/jpeg']

  validates :name,
            :vat,
            :address,
            presence: true

  validates :vat, uniqueness: true
end
```

Codice 11: Classe `Organizer` che utilizza il concern per il `creator`

5.2 Controller

I controller sono le classi che si occupano di rispondere alle chiamate effettuate agli endpoint esposti dalla API. I metodi che gestiscono queste richieste sono chiamati

"action" in Rails. Di seguito viene descritta l'implementazione delle action realizzate nei controller.

5.2.1 Implementazione delle action

Index

Implementa gli endpoint descritti in §4.3.1 e §4.3.7. Verifica che l'utente sia autorizzato ad eseguire questa azione ed effettua il rendering paginato della lista dei record richiesti, dopo che questa è stata filtrata in base ai permessi dell'utente che ha effettuato la richiesta. Serializza i record mostrati, utilizzando un serializzatore conforme a quanto deciso durante la fase di progettazione e descritto in §4.3.1.

Show

Implementa l'endpoint descritto in §4.3.2. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a visualizzarlo e lo restituisce, serializzato in modo conforme a quanto dichiarato nella descrizione dell'endpoint.

Create

Implementa gli endpoint descritti in §4.3.3 e §4.3.8. Controlla che tutti i parametri passati nel corpo della richiesta siano permessi e procede con la creazione del nuovo record. Prima di salvarlo nel database si assicura che l'utente sia autorizzato a creare quel record. Se la creazione avviene all'interno di un altro record, questo viene associato con quello appena creato. Infine associa i file al record, se richiesto e previsto, e lo restituisce serializzato come descritto in §4.3.2.

Update

Implementa l'endpoint descritto in §4.3.4. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a modificarlo. Controlla che tutti i parametri passati nel corpo della richiesta siano permessi e procede con la modifica degli attributi del record, corrispondenti ai parametri passati. Associa i file al record, se richiesto e previsto, infine lo restituisce serializzato come descritto in §4.3.2.

Destroy

Implementa l'endpoint descritto in §4.3.5. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a eliminarlo. Procede con l'eliminazione fisica o logica, dove previsto, e infine lo restituisce serializzato come descritto in §4.3.2.

5.2.2 APIController

`APIController` è la classe base da cui eredita ogni controller della API implementato. Include del codice per gestire automaticamente le eccezioni che possono essere sollevate dal backend e il relativo *rendering* dell'errore. A quelli già presenti ho deciso di aggiungere la gestione delle eccezioni:

- `Pundit::NotAuthorizedError`: sollevata in caso di autorizzazione fallita. Restituisce il codice HTTP `403`;
- `ActionController::UnpermittedParameters`: sollevata nel caso in cui sia presente un parametro non consentito nel corpo di una richiesta. Restituisce il codice HTTP `400`.

Inoltre, ho deciso di rifattorizzare due operazioni che si ripetevano in quasi tutti i controller:

***Attach* dei file** I metodi `create` e `update` accettano un hash (simile a un array associativo di altri linguaggi), quindi è sufficiente passare ai metodi i parametri presenti nel corpo della richiesta, per assegnare il valore dei normali attributi di un record. Gli *attachment* non vengono considerati normali attributi, quindi devono essere associati uno alla volta al record. Questo porta a una grande quantità di codice *boilerplate*, quindi ho creato il metodo `attach_files_to record, from_attributes_in: :attachments`, che associa ogni file contenuto nel parametro della richiesta, specificato con `from_attributes_in`, al record `record`.

***Rendering* della paginazione** Nelle risposte paginate è necessario ritornare, oltre alla lista dei record, anche le informazioni relative al numero totale di pagine e di record disponibili. Tutto questo viene gestito dal metodo `render_pagination collection_hash, each_serializer:`, a cui bisogna passare un hash contenete la lista dei record e un serializzatore, che agisce su ogni record.

5.3 Gestione dei permessi

5.3.1 Policy

Per l'implementazione della gestione dei permessi, è stata utilizzata la gemma Pundit. Utilizza delle policy, dei *plain old ruby object* (PORO), chiamati `ModelPolicy`, per definire le condizioni per cui l'utente è autorizzato a eseguire una specifica azione sul record del modello `Model` interessato. I PORO sono classi che non hanno alcuna dipendenza dal framework o da librerie esterne.

5.3.2 Action

Per autorizzare una specifica action viene utilizzato un metodo con lo stesso nome, seguito dal punto di domanda, che nelle convenzioni di Ruby indica i metodi che ritornano un valore booleano. Questo metodo viene chiamato nel momento in cui, nella action di un controller, viene chiamato il metodo `authorize record`, dove `record` è il record interessato dalla action.

Ad esempio, se il metodo `update?` della policy `PlatformPolicy` ritorna `true`, allora l'utente che intende modificare il record interessato dalla action è autorizzato a farlo. L'implementazione dei metodi delle policy rispetta quanto deciso nella fase di progettazione, descritta in §4.4.2.

5.3.3 Scope

Le action `index` richiedono che la lista dei record da ritornare venga filtrata, restituendo solamente i record che l'utente ha il permesso di visualizzare. Questo è stato

implementato nel metodo `resolve` della classe `Scope`, definita all'interno di ogni policy. Chiamando il metodo `policy_scope` nella action di un controller, passandogli la lista di record da filtrare, si ottiene la lista filtrata.

5.3.4 Parametri permessi

Per implementare le condizioni definite sui parametri permessi in §4.4.3, si definisce il metodo `permitted_attributes` all'interno della policy, che deve restituire sempre un array di simboli, rappresentanti i parametri permessi. Questo metodo, chiamato nella action di un controller, restituisce i parametri presenti nel corpo della richiesta, se sono tutti permessi, altrimenti solleva un'eccezione.

5.4 Test di unità

Durante la codifica dei modelli sono stati definiti alcuni test di unità, prevalentemente sulle validazioni dei dati. Non sono stati testati tutti i modelli e, in particolare nessun controller, perché non era un'attività richiesta esplicitamente dal committente, quindi mi è stato chiesto dal *project manager* di investirci una quantità limitata di tempo. I test definiti sono stati implementati utilizzando la gemma RSpec, che fornisce strumenti per il *behaviour-driven development* (BDD), come da prassi aziendale di Moku. La pratica del BDD riprende molti concetti del *test-driven development*, prestando particolare attenzione alla leggibilità dei test e alla loro intuitività. RSpec, in particolare, utilizza diverse *keyword* che puntano a rendere il codice quanto più possibile simile al linguaggio naturale, gli stessi nomi dei test sono spesso delle vere e proprie frasi. Un esempio di un test implementato è:

```
it 'is not valid without a role' do
  subject.role = nil
  expect(subject).not_to be_valid
end
```

Codice 12: Esempio di un test implementato con RSpec

Sono stati implementati test solamente per i seguenti modelli, per un totale di 36 test:

- User;
- Platform;
- Organizer.

In particolare, i test scritti per il modello degli utenti sono i più esaustivi, perché sono quelli su cui è stato dedicato più tempo e comprendono test sulle validazioni, su alcuni aspetti dell'autenticazione e sull'associazione con l'utente `creator`.

Capitolo 6

Conclusioni

6.1 Raggiungimento dei requisiti

La tabella seguente riporta lo stato di completamento e soddisfazione dei requisiti, definiti in §2.2.

Requisito	**Importanza**	**Stato**
Gestione e pianificazione del progetto attraverso kanban board condivisa	Obbligatorio	Rispettato
Analisi dei flussi attuali e delle API richieste	Obbligatorio	Rispettato
Progettazione ed implementazione dei modelli e dei controller, a partire dai requisiti raccolti	Obbligatorio	Rispettato
Analisi ed integrazione Zoom, GoToWebinar, Webex	Obbligatorio	Non rispettato
Coordinamento con il cliente finale	Desiderabile	Rispettato
Integrazione team	Desiderabile	Rispettato
Integrazione stampante biglietti	Desiderabile	Non rispettato
Suite di testing del software prodotto	Desiderabile	Non completamente rispettato
Documentazione completa	Desiderabile	Non rispettato
Ulteriori modifiche all'applicazione che esulano da quando riportato nel piano di lavoro	Opzionale	Rispettato

Tabella 6.1: Tabella dello stato di completamento dei requisiti

Come già spiegato in §2.2, il conseguimento dei requisiti, in alcuni casi, è stato dipendente da fattori esterni:

- la suite di testing è stata iniziata, ma, dopo la richiesta da parte del *Project*

Manager di dedicarci una quantità di tempo limitata, l'esaustività e la completezza dei test si è ridotta, per questo il requisito è stato contrassegnato come "Non completamente rispettato";

- le comunicazioni con il committente sono state spesso soggette a rallentamenti notevoli. Ad esempio, durante tutta la durata dello sprint 5, l'azienda che ha commissionato il progetto si è dedicata completamente ad altri progetti, sospendendo le comunicazioni e l'avanzamento dei lavori sul progetto di Evvvents. Tra le varie domande sulle scelte da perseguire nel progetto ne erano previste alcune sullo scopo e le funzionalità che avrebbero dovuto offrire le integrazioni. Non essendo stato possibile definire questi dettagli insieme al committente, entro il periodo dello stage, le integrazioni non sono state implementate, quindi il requisito obbligatorio non è stato implementato. Nonostante questo lo stage è stato considerato concluso con successo.

Nonostante il mio stage sia giunto al termine, il progetto Evvvents rimane ancora in corso di sviluppo, con diversi aspetti da completare e implementare, data la sua grandezza e complessità. Il lavoro verrà portato avanti da un altro stagista, che prenderà il mio posto.

6.2 Valutazione personale

L'esperienza di stage nel suo complesso è stata un'occasione per mettere alla prova tutte le competenze e le conoscenze fornite dai corsi seguiti in questi anni. Ho avuto modo di sperimentare le mie abilità nel progettare un sistema complesso, partendo dalla ridefinizione dello strato di persistenza, passando per la progettazione delle componenti e delle classi, utilizzando una combinazione strutturata di concetti e insegnamenti, sia pratici che teorici, appresi durante tutto il corso di laurea.
Lo svolgimento dello stage ha avuto modo di lasciarmi diversi insegnamenti:

- ho imparato un nuovo linguaggio di programmazione, che ha uno stile diverso dai linguaggi che avevo usato fino a questo momento e che mi ha fatto scoprire l'applicazione di paradigmi e pratiche a me nuove;
- ho imparato la struttura e la filosofia del framework Ruby on Rails, apprezzando (con un occhio critico) le sue convenzioni. Mi ha permesso di consolidare le mie conoscenze sull'applicazione del pattern MVC nello sviluppo di applicazioni web;
- ho notato ancora di più quanto sia importante riferirsi alla documentazione ufficiale del software utilizzato e di fonti affidabili, anche se questo può richiedere un certo tempo e fatica;
- ho scoperto l'ambiente lavorativo aziendale, con tutte le dinamiche che ne derivano, i ruoli del personale, le convenzioni e le metodologie seguite, oltre agli specifici strumenti utilizzati per la gestione dei progetti o della comunicazione aziendale, come Jira, Confluence o Slack;
- ho avuto conferma dell'importanza e del piacere di un confronto con i colleghi, per chiedere o risolvere dubbi e collaborare alla soluzione di un problema.

Nell'affrontare un'esperienza lavorativa concreta, ho avuto conferma dell'utilità delle varie competenze tecniche che ho appreso grazie allo svolgimento dei progetti, degli approfondimenti personali, svolti parallelamente allo studio e delle competenze trasversali

acquisite tramite esperienze esterne. Sono tutte abilità che si potrebbero definire "di contorno", ma che permettono di risolvere problemi e difficoltà che si possono presentare durante il mio lavoro, o quello dei miei colleghi.

Appendice A

Diagramma ER del sistema

Di seguito viene riportato il diagramma ER completo dell'applicazione. Il diagramma è stato creato con l'intenzione di essere una guida per la creazione dei modelli, quindi punta ad essere un riflesso di quanto è stato e verrà implementato, invece che una rappresentazione dettagliata della struttura del database. Questo implica che alcuni dettagli non sono esplicitati nel diagramma, perché considerati superflui (perché implementati dal framework) o già presenti nel sistema.
Sono state adottate le seguenti convenzioni:

- tutte le entità che possiedono l'attributo `creator_id` hanno implicitamente una relazione del tipo (1, N) con l'entità `users`. La relazione non è stata rappresentata perché avrebbe aumentato notevolmente la complessità del diagramma, riducendone la leggibilità, al netto di una semplificazione ritenuta sufficientemente intuitiva;
- la colonna sinistra delle tabelle indica eventuali proprietà particolari degli attributi:
  - N: l'attributo può assumere il valore `NULL`. Di conseguenza si assume che tutti gli attributi che non possiedono questa proprietà vengono considerati `NOT NULL`;
  - U: l'attributo possiede un indice di unicità, all'interno dell'entità, cioè viene considerato `unique`;
  - FK: indica gli attributi che sono chiavi esterne (*foreign key*) di altre entità. Non viene specificata l'entità a cui si riferiscono, perché inferita dal nome dell'attributo, come da convenzioni di Rails. Ad eccezione di alcuni casi, comunque ritenuti intuitivi, tutte le chiavi esterne si chiamano *`entity`*`_id`.
- la colonna destra delle tabelle indica il tipo dell'attributo utilizzato nella migrazione. L'unica eccezione è il tipo fittizio `BLOB`, che indica che quell'attributo è in realtà un attachment, dichiarato a modello.

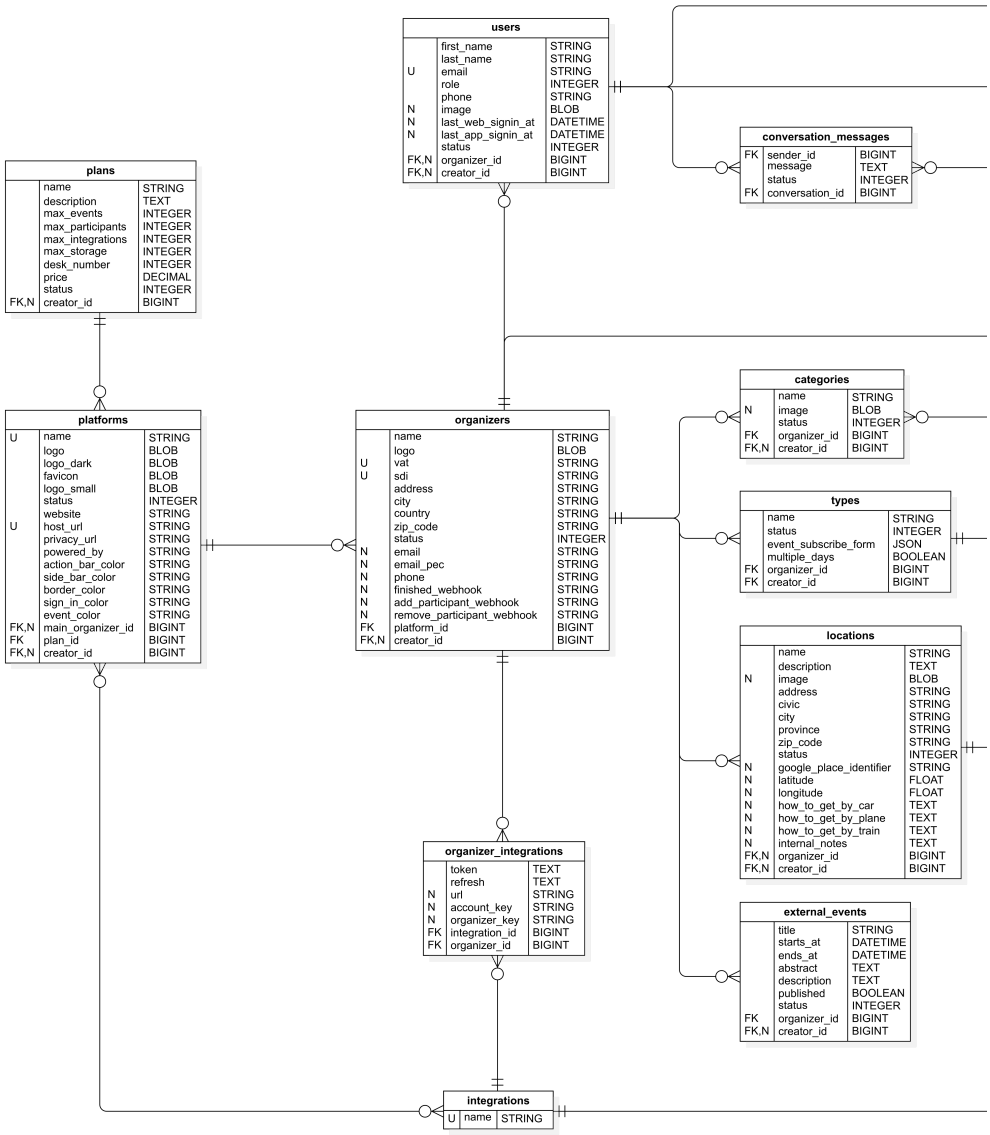

Figura A.1: Sezione sinistra del dettaglio del diagramma ER

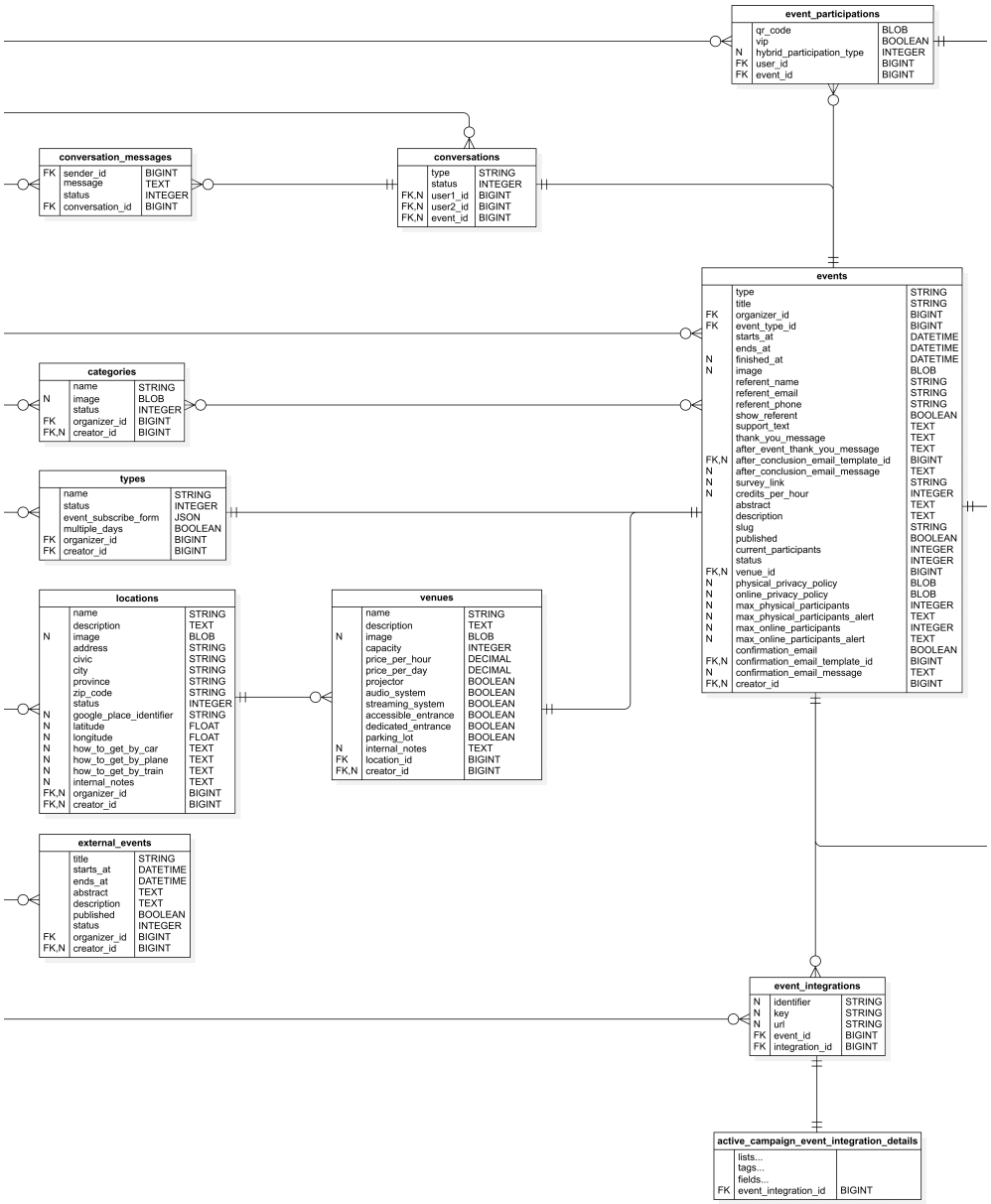

Figura A.2: Sezione centrale del dettaglio del diagramma ER

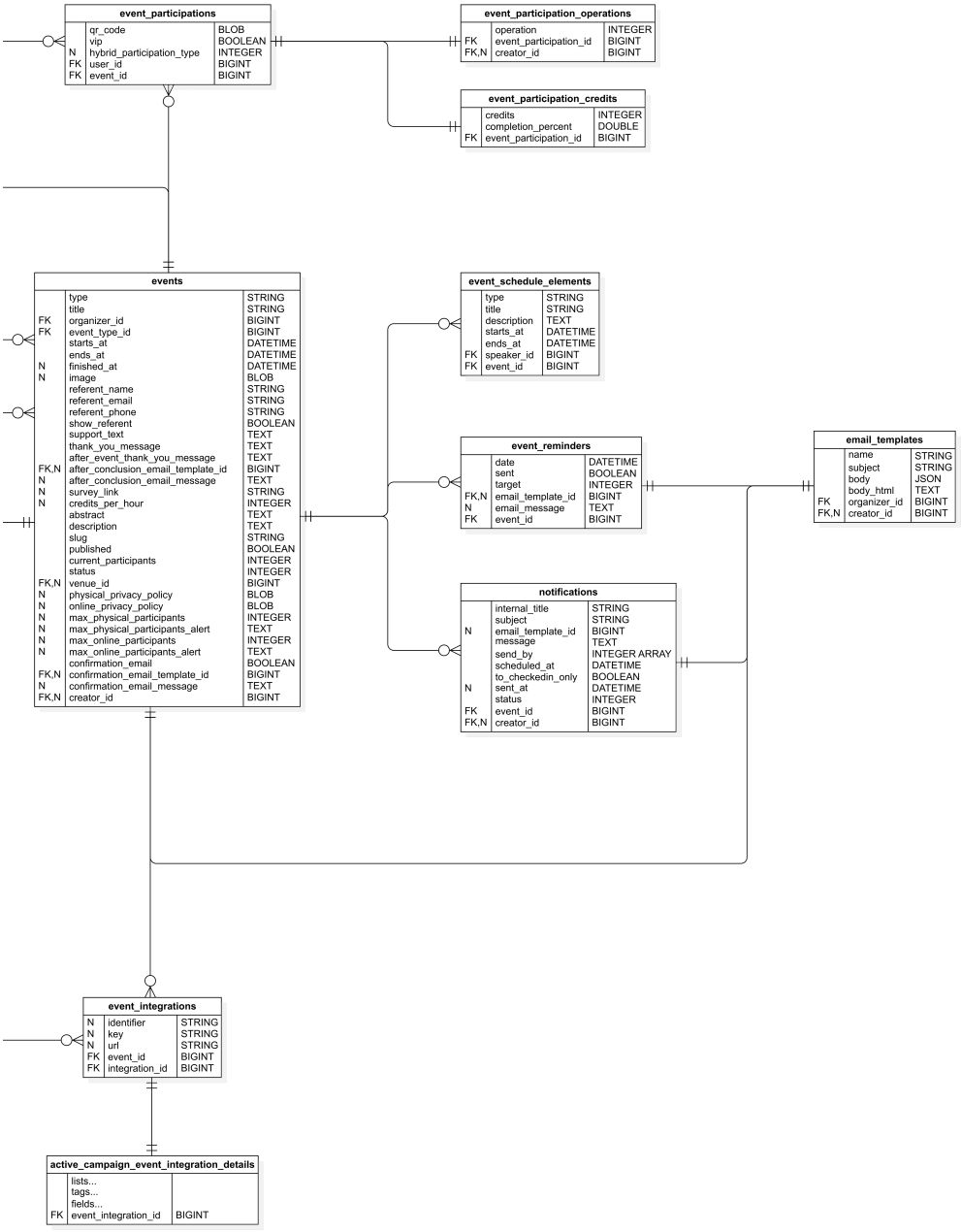

Figura A.3: Sezione destra del dettaglio del diagramma ER

Bibliografia

Siti web consultati

Documentazione ufficiale di Devise. URL: https://rubydoc.info/github/heartcombo/devise/.

Documentazione ufficiale di Devise Token Auth. URL: https://devise-token-auth.gitbook.io/devise-token-auth/.

Documentazione ufficiale di PostgreSQL. URL: https://www.postgresql.org/docs/.

Documentazione ufficiale di Pundit. URL: https://www.rubydoc.info/gems/pundit.

Documentazione ufficiale di Ruby. URL: https://ruby-doc.org/.

Documentazione ufficiale di Ruby on Rails - metodo `add_column`. URL: https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html#method-i-add_column.

Doyle, Kerry. *Programming in Ruby: A critical look at the pros and cons*. URL: https://www.techtarget.com/searchapparchitecture/tip/Programming-in-Ruby-A-critical-look-at-the-pros-and-cons.

Guide ufficiali di Ruby on Rails. URL: https://guides.rubyonrails.org/.

Hartl, Michael. *Ruby on Rails Tutorial: Learn Web Development with Rails*. URL: https://www.railstutorial.org/book.

Ken Schwaber, Jeff Sutherland. *The 2020 Scrum Guide*. URL: https://scrumguides.org/scrum-guide.html.

Sito ufficiale di Ruby. URL: https://www.ruby-lang.org/.

Sito ufficiale di Ruby on Rails. URL: https://rubyonrails.org/.

Stack Overflow. URL: https://stackoverflow.com/.