ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

PROGETTAZIONE E SVILUPPO DI UN SISTEMA DI SCHEMA MANAGEMENT PER LA DEFINIZIONE DELLE METRICHE GENERATE DA APPLICAZIONI MOBILE

Elaborato in
SVILUPPO DI SISTEMI SOFTWARE

Relatore
Prof. MIRKO VIROLI

Co-relatore
Dott. DANIELA GHIRONI

Presentata da
ELIA DI PASQUALE

Terza Sessione di Laurea
Anno Accademico 2018 – 2019

PAROLE CHIAVE

Schema Management

Applicazioni Mobile

Metriche

Eventi

Schema Registry

La forza di una nazione deriva dall'integrità della casa
- Confucio

Alla mia famiglia

Indice

Introduzione

I dati stanno diventando ogni giorno più vitali per le aziende. Analizzando i dati, è possibile ottenere approfondimenti, valutare alternative e creare strategie di business di successo (ad esempio, eseguire A/B testing, analizzare campagne di marketing, prevedere lo user lifetime value e così via). Adottare un processo decisionale data-driven significa operare verso obiettivi aziendali chiave sfruttando dati verificati ed analizzati anziché basarsi su sensazioni o opinioni non supportate da dati empirici.

L'aumento di dati schema-free ha portato a mettere in discussione il valore e la necessità degli schemi. In contesti specifici gli schemi possono sembrare troppo restrittivi ed è stato ipotizzato che lo sviluppo software possa essere più veloce e più agile senza di essi. Tuttavia, solo perché è possibile procedere senza schemi non significa che sia saggio farlo. Non avere una modalità standardizzata per definire i dati all'interno di un'organizzazione può portare a seri problemi legati alla produzione, al consumo e all'evoluzione dei dati stessi. Può essere una situazione accettabile nel breve termine, ma porterebbe ad inutili difficoltà ed è una realtà non scalabile se si immagina di estenderla a potenzialmente migliaia di differenti categorie di dati. Gestire i dati attraverso schemi potrebbe comportare un costo iniziale, ma il vantaggio è enorme e permanente. Uno schema fa riferimento all'organizzazione delle informazioni come fosse un modello di come sono strutturati i dati, il che è utile per l'organizzazione e per l'interpretazione delle informazioni. Gli schemi ci consentono di avere una via diretta per l'interpretazione della grande quantità di informazioni disponibili nel nostro sistema; in questo modo informazioni, formati e simboli per le espressioni sono compresi universalmente, eliminando la possibilità di confusione.

Questo progetto nasce all'interno di **Bending Spoons**, una tech company in rapida crescita focalizzata sulla progettazione e commercializzazione di applicazioni mobile. Nonostante sia un'azienda così giovane, hanno già ottenuto risultati notevoli, con oltre 270 milioni di download delle loro app e milioni di utenti attivi ogni mese.

L'obiettivo principale di questo progetto è quello di regolare le comunica-

zioni e la struttura degli eventi generati, ed il loro invio ai sistemi dedicati; nella prima versione ci concentreremo sugli eventi generati dalle applicazioni, ma questo sistema può essere utilizzato anche per i dati inviati da altri servizi. In particolare, lo scopo del progetto è quello di standardizzare la struttura degli eventi, attraverso la definizione e l'utilizzo di **schemi di eventi**, e memorizzare ed organizzare l'evoluzione e la distribuzione di questi schemi attraverso un sistema centralizzato, di seguito denominato **schema registry**.

La prima fase che verrà attuata sarà quella di analisi e revisione dei processi aziendali esistenti, in modo da comprendere al meglio quali sono le esigenze e i problemi delle metodologie attuali, così da poter strutturare un sistema in grado di migliorare, e al tempo stesso di non stravolgere, quelle che sono le dinamiche correnti. La definizione e lo sviluppo di un sistema di tale complessità e ampiezza rendono necessarie precise tecniche di ingegneria del software, in modo da poter garantire caratteristiche essenziali quali correttezza, modularità e scalabilità. Altri aspetti importanti per questo genere di prodotto sono un alto livello di testabilità e manutenibilità, caratteristiche che rendono il sistema facilmente operabile in caso di problemi, nonché alla base per sviluppi ed espansioni future.

Questo documento è stato suddiviso in cinque capitoli che riassumono le varie fasi del progetto.

Il primo capitolo contiene un'introduzione ai sistemi di schema registry e alle logiche di serializzazione; viene quindi descritto il problema affrontato, nonché i diversi sistemi già esistenti in letteratura.

Nel secondo capitolo viene descritto il sistema corrente analizzandone il funzionamento e le criticità; verranno quindi presentati i diversi requisiti per lo sviluppo del progetto e discussi i contesti di utilizzo.

Nel terzo capitolo viene descritta la fase di progettazione software del sistema, avvalendosi di diagrammi UML per la definizione dei principali meccanismi di funzionamento.

Il quarto capitolo è dedicato alle scelte implementative derivate dall'analisi precedente, nonché alle tecnologie utilizzate per l'implementazione delle principali sotto parti del sistema.

Il quinto capitolo descrive le fasi di test e vengono inoltre analizzati i risultati ottenuti e l'usabilità del sistema.

Seguono conclusioni e ringraziamenti.

Capitolo 1

Analisi

1.1 Valutazione del sistema attuale

Per comprendere al meglio quali siano le caratteristiche utili del sistema che si andrà a progettare è stata effettuata una fase di raccolta dei requisiti per conoscere lo stato attuale dei processi di governo e le loro principali problematiche.

Questa fase di analisi del sistema attuale e di raccolta delle informazioni riguardanti le modalità di utilizzo del sistema è un passaggio fondamentale per il progetto stesso, in quanto durante questa fase non si analizzano solo le informazioni e i modi in cui vengono attualmente gestite, ma anche i processi stessi che operano a livello aziendale; è proprio all'interno di questi processi che il nuovo sistema si dovrà poi inserire, in modo da modificarli in meglio in termini di usabilità e stabilità.

Per questo durante questa attività sono stati interrogati, in modalità di intervista, i responsabili dei principali gruppi di lavoro che sono coinvolti nell'attuale processo di utilizzo degli elementi in analisi; questi team saranno poi gli attori principali che utilizzeranno attivamente il nuovo sistema.

1.1.1 Panoramica dei team

Vengono di seguito riportati i diversi team che sono coinvolti nelle dinamiche di gestione ed utilizzo delle informazioni gestite dal sistema:

- **product team**: sono coloro che si occupano di gestire la progettazione e lo sviluppo di ogni applicazione. Prendono decisioni in termini di contenuti, dinamiche di utilizzo e funzionalità; si coordinano inoltre con gli altri team in modo da definire deadline di sviluppo, attività di marketing, ecc. Di particolare interesse per l'interazione con il sistema è l'attività

di definizione delle metriche da tracciare per ogni applicazione che viene costruita da zero o semplicemente aggiornata;

- **data analysis team**: i componenti di questo team si occupano principalmente della gestione e dell'analisi di tutto ciò che è legato ai dati generati dalle varie applicazioni. In particolare utilizzano i dati raccolti dai componenti del sistema (presentanti nella sezione seguente) e svolgono diverse tipologie di analisi per svariati scopi (marketing, dell'applicazione a scopo di miglioramento delle stesse ed identificazione dei punti critici, ecc.). Si relazionano inoltre con il product team in modo da offrire indicazioni sui parametri di interesse per le varie applicazioni;

- **software engineers team**: team composto da software engineers che progettano e sviluppano l'applicativo vero e proprio. Sulle indicazioni del product team gestiscono tutto il flusso di costruzione dell'applicazione; al tempo stesso vengono utilizzate le informazioni definite circa gli eventi da tracciare per comporre il relativo modulo a livello applicativo.

1.1.2 Panoramica dei componenti

Di seguito sono descritti alcuni dei componenti che vengono utilizzati nei sistemi aziendali, e che andranno ad interagire anche con quanto di nuovo verrà introdotto e sviluppato:

- **Pico**: è lo strumento interno per ricevere e archiviare eventi generati da tutte le applicazioni e dai servizi di backend. I produttori inviano gli eventi al server Pico contattando gli endpoint esposti, quindi Pico convalida e pre elabora gli eventi ricevuti prima di memorizzarli su BigQuery;

- **BigQuery**: BigQuery è un data warehouse cloud server-less, altamente scalabile e con BI Engine un in-memory e sistemi di machine learning integrati; è un prodotto di Google Cloud Platform [2]. Offre agli utenti la possibilità di gestire i dati utilizzando query di tipo SQL per analisi in tempo reale e consente l'analisi interattiva di enormi set di dati; infatti viene utilizzato per gestire o analizzare big data.

  Con BigQuery è possibile analizzare tutti i batch e gli stream di dati creando un data warehouse logic, nonché dati da oggetti di archiviazione e fogli di calcolo;

- **Ambrogio**: Ambrogio è la piattaforma personalizzata di gestione del ciclo di vita delle applicazioni per gestire l'intero processo di sviluppo e

rilascio delle app. Grazie ad Ambrogio, è possibile evitare innumerevoli attività di routine manuali, come la creazione di profili di provisioning, il caricamento di materiale promozionale per decine di lingue diverse, l'esecuzione di test automatici e la preparazione di file binari sia per i test che per l'invio all'App Store.

1.1.3 Eventi

Gli elementi chiave dell'analisi aziendale sono gli eventi gestiti dal sistema attuale. In particolare, esistono sia eventi generati dagli utenti che eventi generati dai servizi di backend; questa analisi si concentra sulla prima categoria.

Gli eventi generati dalle applicazioni vengono salvati in BigQuery per essere successivamente elaborati ed analizzati.

Per ciascun tipo di evento ricevuto da Pico, viene automaticamente creata una tabella dedicata. Gli eventi comuni a tutte le app sono gli eventi session, install e user action:

- **Install**: evento generato quando l'utente installa o aggiorna una delle applicazioni;
- **Session**: evento generato quando l'utente avvia l'applicazione e la utilizza;
- **User action**: evento generato quando l'utente esegue un'azione specifica sull'applicazione (ad es. Il completamento di un allenamento in un'app di fitness)

I product lead definiscono nuovi eventi user action utilizzando un foglio di calcolo, e la prima versione di questo sistema si concentra nel fornire uno strumento per definire lo schema delle possibili azioni eseguite dall'utente all'interno dell'applicazione.

Data la natura statica della struttura delle prime due tabelle, che modellano eventi e informazioni precise, queste non richiedono un meccanismo di evoluzione; il focus è quindi sulla tabella di user action, dove vengono salvati tutti gli eventi e le informazioni che non riguardano l'installazione e le sessioni degli utenti.

Esiste un particolare set di informazioni in tutti i tipi di eventi (ad es. install, session, ecc.) che si chiama user info. È composto sia da campi generali (ad es. paese, lingua, ecc.) che da una serie di campi dipendenti dall'applicazione.

A differenza degli altri dati, non rappresentano eventi o azioni, ma modellano informazioni relative all'utente; ci sono alcune informazioni standard che vengono sempre monitorate (paese, dispositivo, lingua, ec.) che sono riportate anche nelle varie tabelle sopra elencate, ma ci sono anche alcune informazioni personalizzate per ogni applicazione (come specificato sopra). Una volta che queste informazioni sono state rilevate, verranno replicate ogni volta che viene tracciato qualsiasi evento correlato all'utente.

Dato il loro particolare comportamento, la loro definizione è inizialmente esclusa dalla funzionalità del sistema.

1.1.4 Foglio di calcolo "User Action"

I product lead e i data analyst condividono le informazioni attraverso un foglio di calcolo che definisce la struttura delle azioni eseguite dall'utente in un'applicazione specifica.

Di seguito sono riportate alcune parti di esempio di un foglio di calcolo:

Ogni team di prodotto utilizza una struttura differente, ma in generale i campi principali sono i seguenti:

- **section**: aggruppamento concettuale di action kind in categorie (utilizzate solo per la leggibilità del documento);
- **action kind**: nome (identificatore) scelto come riferimento per l'azione che verrà tracciata;
- **action description**: informazioni che descrivono l'evento con maggior dettaglio.

Ad ogni*action kind* corrispondono 0-n istanze delle informazioni seguenti:

- **action info**: sottocampo di un' *action kind* che descrive una specifica funzionalità della stessa (es. per l'action kind "element_deleted", che viene innescata quando un utente cancello un elemento, un'utile action info potrebbe essere "element_type", che indicherebbe il tipo di elemento eliminato);
- **data type**: tipo di dato che deve essere tracciato;
- **description**: informazioni sul parametro in questione;
- **allowed values and examples**: valori ammessi ed esempi di utilizzo;

	SECTION	ACTION KIND	ACTION DESCRIPTION	ACTION INFO	DATA AVAILABLE FROM *min build number*
Correctly implemented	App setup	app_setup_started	AppSetup starts.		1048
Correctly implemented		app_setup_error	First *AppSetup.LibraryError* during AppSetup fires.	app_setup_error_type	1048
Correctly implemented				app_setup_network_error_code	1048
Correctly implemented				app_setup_server_error_code	1048
Correctly implemented				app_setup_generic_error_description	1048
Correctly implemented		app_setup_completed	AppSetup is completed successfully.		1048
Implemented, not tested	Legal	legal_screen_accepted	User successfully accepts TOS/PN in the TOS screen by tapping on Continue	old_tos_version	
Implemented, not tested				new_tos_version	
Implemented, not tested				old_pn_version	
Implemented, not tested				new_pn_version	
Implemented, not tested		legal_screen_displayed	The TOS/PN screen is displayed	old_tos_version	
Implemented, not tested				new_tos_version	
Implemented, not tested				old_pn_version	
Implemented, not tested				new_pn_version	
Implemented, not tested		legal_screen_error	There is an error when showing the TOS/PN screen	error_code	
Implemented, not tested		paywall_demo_popup	Everytime the user taps on a button in the pop-up with the gift in the demo paywall case.	demo_free_export	1052
Implemented, not tested				demo_unlimited_exports	1052
Implemented, not tested		paywall_demo_banner	Out of exports banner is displayed to the user in case of the demo paywall	banner_type	1052
Not yet implemented		paywall_pro_banner	Pro banner is displayed to the user in case of the soft paywall with pro features		

ACTION INFO	DATA TYPE	ALLOWED VALUES and EXAMPLES	DESCRIPTION
action_undone	bool	0, 1	Whether or not the user undid the action that overrode the animation on their clip
app_setup_error_type	string	"content_error", "io_error", "network_error", "server_error", "generic_error"	Type of *AppSetup.LibraryError*.
app_setup_generic_error_description	string	*e.g. "A custom description of the error"*	Generic error description.
app_setup_network_error_code	integer	... -1, 0, 1, ...	Network error code.
app_setup_server_error_code	integer	... -1, 0, 1, ...	Server error code.
apply_all_change	string	"speed", "volume", "duration, "filter", "adjustments", "background", "color", "font", "alignment", "opacity", "transition_type", "animation"	Type of change the that was applied to all the elements of the same kind.
apply_all_element	string	"photo", "video", "title", "music", "sfx", "rec", "text", "transition"	Type of element the "apply to all" has interacted with.
banner_type	string	"checkpoint_triggered", "project_setup", "main_editor", "at_export"	The type of banner shown, same as settings
catalog_source	string	"add", "change"	Action that triggered the event.

Figura 1.1: Frammento di un foglio di calcolo per la gestione degli eventi di un'applicazione

- **data available from**: *build number* dell'applicazione [1] dal quale tracciare l' *action info*;

- **status**: indica lo stato delle varie *action info*:

  - Not yet implemented
  - Implemented, not tested
  - Correctly implemented
  - Work in progress
  - Incorrectly implemented

1.1.5 Workflow

Il processo attuale è descritto di seguito (e dalla figura 1.2):

- i product lead definiscono le strutture degli eventi relativi ad un'applicazione che sta per essere sviluppata/ aggiornata riportando le informazioni in un foglio di calcolo;

- dopo aver letto le specifiche, un software engineer si occupa di definire, lato applicazione, le strutture degli eventi e la logica della loro generazione;

- una volta che lo sviluppo è stato completato e l'applicazione rilasciata, questa invia i nuovi eventi a Pico;

- Pico effettua dei controlli sommari di validazione sulla struttura degli eventi basati su schemi standard contenuti all'interno di Pico stesso, dopodiché salva gli eventi validi su BigQuery;

- i data analysts analizzano i dati presenti in BigQuery e utilizzano il foglio di calcolo creato dal product lead per comprendere le informazioni salvate nelle tabelle.

[1] build number: codice alfanumerico associato ad una specifica release della relativa applicazione

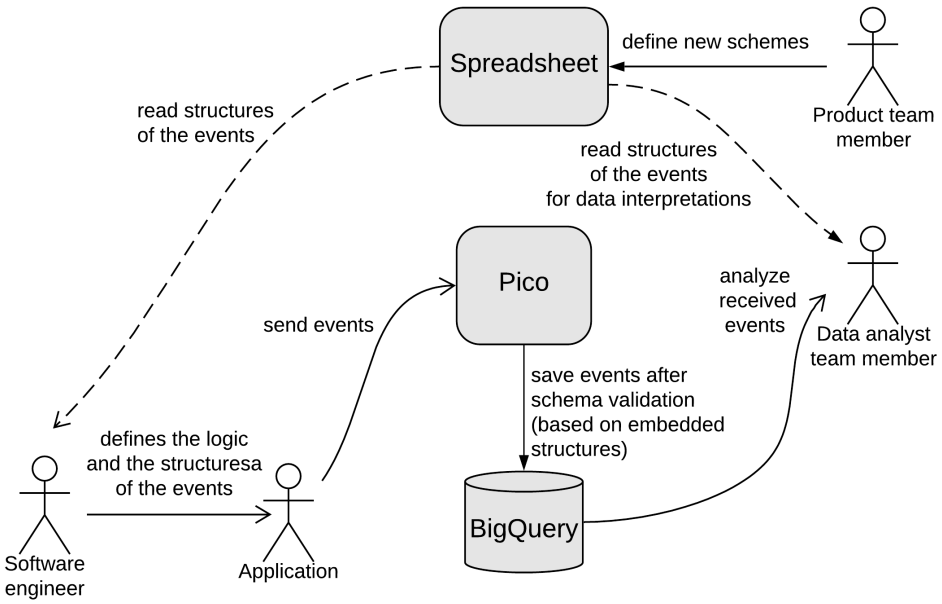

Figura 1.2: Schema di funzionamento attuale del processo di gestione degli eventi.

1.1.6 Problemi attuali

Dalla fase di raccolta delle informazioni, sono stati identificati i seguenti problemi:

- developer che, con o senza l'accordo del responsabile del prodotto, tracciano le azioni pertinenti a fini di analisi ma che non sono riportate nel documento;
- eventi aventi *action info* diversi da quelli definiti nel foglio di calcolo (a volte perché i nomi sono stati cambiati dagli sviluppatori, a volte perché ci sono alcuni errori di battitura);
- documentazione degli user event mancante o incompleta;
- i campi *action info* che fanno riferimento ad *action kind* differenti hanno lo stesso nome e, di conseguenza, finiscono nella stessa colonna della tabella BigQuery;
  - questo di solito non accade alla prima versione perché, in quella situazione, le metriche sono definite tutte insieme (e quindi coloro che le definiscono noterebbero delle equivalenze tra i nomi). È quindi più probabile che si verifichi nelle fasi di aggiornamento.

- non esiste uno standard nella creazione di fogli di calcolo (ogni product lead può decidere di utilizzare la propria convenzione);
- alcuni fogli di calcolo di prodotto non sono aggiornati o non sono stati costruiti correttamente;
- campi con lo stesso significato chiamati in modi diversi, a seconda dello sviluppatore che li scrive (rende difficile l'analisi dei dati);
  - (es. utilizzare, in un'app di fitness, "workout_id" per identificare un allenamento in un evento specifico e "id_workout" in un altro)
- il campo "data available from" indica solo quando sono stati aggiunti eventi, non supporta l'eventuale deprecamento degli eventi nelle build successive;
- dato un action info che può assumere un intervallo di valori specifico e chiuso (ad esempio, per il campo "photo_from" i valori possibili potrebbero essere "camera", "gallery", "shared".), è impossibile tenere traccia di eventuali aggiunte/rimozioni a questo insieme di valori.

1.2 Goals

Una volta che sono state chiarite le dinamiche attuali e i relativi problemi più nel dettaglio, è possibile reiterare i golas relativi al progetto in modo da identificarli con maggior precisione.

Il progetto mira a:

- definire un approccio standard per rappresentare gli eventi che sono gli oggetti delle comunicazioni, al fine di definire rigorosamente la loro struttura nel dettaglio. L'approccio definito deve essere abbastanza flessibile da soddisfare le esigenze di evoluzione degli schemi;
- gestire la creazione e l'archiviazione strutturata di questi schemi in modo che possano essere consultati e recuperati da altri servizi/componenti del sistema (sia durante il processo di codifica delle informazioni che durante la verifica della loro struttura).

  In particolare, utilizzare la conoscenza centralizzata degli schemi per:

  - automatizzare parte del processo di definizione degli eventi da tracciare lato applicazione con l'obiettivo principale di ridurre il più possibile gli errori;

- verificare la correttezza degli eventi durante il loro salvataggio;
- consentire la convalida degli eventi in entrata rispetto allo schema fornito;
- servire come *live documentation* per tutte le persone coinvolte nei processi di ingestione e analisi dei dati.

1.3 Requisiti

In risposta alle necessità individuate e al perseguimento degli obiettivi sopra descritti, sono stati identificati i seguenti requisiti.

1.3.1 Funzionali

- Il sistema deve consentire ai product lead di definire nuovi schemi per gli eventi, inserendo le informazioni relative all'evento stesso e alla sua struttura;
- durante la creazione di uno schema il sistema deve consentire di stabilire la relazione tra lo schema stesso e l'applicazione alla quale verrà applicato;
- il sistema dovrà fornire ad un utente generico un modo per accedere agli schemi oltre che alle informazioni correlate (descrizione, campi, ecc.);
- il sistema deve garantire la fruizione delle informazioni in un formato di serializzazione strutturato, in modo che queste possano essere utilizzate da altri componenti del sistema stesso;
- l'utente deve avere la possibilità di visualizzare gli stati precedenti di un evento modificato, in modo da avere una visione chiara di quali siano le differenze tra le diverse versioni;
- l'utente specializzato deve essere in grado di modificare le informazioni inserite nel sistema, avendo la possibilità di modificare solo i valori che si prestano a modifiche dopo il primo inserimento;
- il sistema deve permettere la possibilità di deprecare un evento (ovvero indicare che quell'evento non è più di interesse per le successive versioni di una data applicazione);

- il sistema deve permettere, in riferimento ad una singola applicazione, di generare il codice per il tracciamento degli eventi legati alla suddetta applicazione, in formato tale che questo possa essere direttamente integrato nel codice stesso che comporrà l'applicazione vera e propria;

- il sistema deve iniettare la definizione dello schema nelle build dell'applicazione, sia nella fase di creazione dell'applicazione che nelle fasi di aggiornamento/revamp:

- il sistema deve permettere l'utilizzo degli schemi salvati per la validazione di eventi inviati dalle applicazioni ai sistemi centrali, in modo da verificarne struttura e parte dei contenuti.

1.3.2 Non funzionali

- Il sistema deve convalidare nuovi schemi ed imporre la compatibilità, verificando la struttura di un dato schema perché questa sia in linea con le regole che verranno definite in fase di design della struttura stessa del sistema;

- il sistema deve impedire modifiche alla struttura dello schema in aggiunta/rimozione di informazioni non ritenute compatibili con l'attuale struttura dello schema, seguendo regole che verranno definite in fase di design della struttura stessa del sistema;

- il sistema deve mantenere un versionamento immutabile degli schemi, in modo che queste siano singolarmente accessibili e mantengano i riferimenti alle strutture originali;

- il sistema deve gestire l'evoluzione degli schemi mantenendo le informazioni relative all'inizio e alla fine del monitoraggio di particolari action kind/action info;

- il sistema deve essere abbastanza flessibile da supportare formati di serializzazione multipli;

- il sistema deve disporre di un meccanismo di autorizzazione e di autenticazione.

1.3.3 Tecnici

Per rimanere conformi allo stack tecnologico e alle best practices di sicurezza adottate in *Bending Spoons*:

- qualsiasi funzionalità di back-end verrà sviluppata in *Python 3.6+*;
- qualsiasi applicazione web front-end verrà sviluppata con *React / Redux / Typescript*;
- qualsiasi codice relativo all'applicazione verrà generato in *Swift*;
- qualsiasi componente verrà distribuito su *Google Cloud Platform* [3].

Capitolo 2

Progettazione del sistema

Nel seguente capitolo verranno descritte le scelte progettuali derivate dall'analisi precedentemente effettuata; verranno quindi definite le caratteristiche dell'architettura generale e delle sottoparti del sistema, corredate da diagrammi UML.

Le diverse fasi progettuali saranno strutturate in modo tale da risultare più astratte possibile rispetto al restante sistema, per facilitare il riutilizzo delle logiche di progetto.

2.1 Architettura generale

Sulla base dei requisiti progettuali che sono stati analizzati precedentemente (vedi sezione 1.3), possiamo ora definire quella che sarà la struttura generale del sistema; queste specifiche saranno poi utilizzate nella fase di implementazione.

2.1.1 Scheletro del sistema

A seguito dell'analisi, è stata pensata una possibile soluzione ed illustrata nei diagrammi seguenti; i componenti verdi sono quelli che verranno aggiunti all'infrastruttura esistente, di cui fanno parte i componenti grigi.

I diagrammi mostrano anche le interazioni con il sistema eseguite dagli attori principali.

Lo schema registry è responsabile della memorizzazione e del recupero dei vari schemi.

Utilizzando API specifiche, questi schemi saranno definiti ed utilizzati da una serie di componenti esistenti (come *Pico* e *Ambrogio*, sezione 1.1.2). Sarà inoltre sviluppata anche un'interfaccia web che comunicherà con lo schema

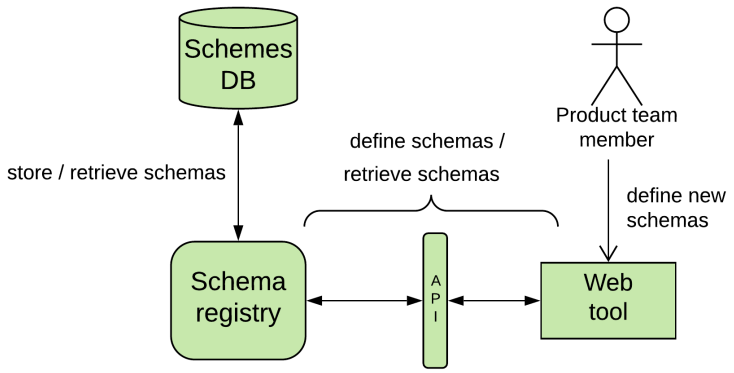

Figura 2.1: Diagramma che illustra le modalità di utilizzo del nuovo sistema da parte dei product lead, in fase di definizione e consultazione degli schemi.

registry, per consentire ai product lead di definire e visualizzare gli schemi; questa interfaccia verrà utilizzata anche dai data analyst per interpretare i dati.

I product lead definiscono, tramite il web tool, gli schemi di eventi relativi ad un'applicazione in fase di sviluppo/aggiornamento. Questi schemi vengono quindi salvati dallo schema registry nel relativo database (figura 2.1).

Ambrogio recupererà gli schemi e li inietterà nell'applicazione in fase di sviluppo; un software engineer dovrà quindi solamente implementare la logica degli eventi iniettati e popolare i relativi sottocampi correlati.

Quando l'app verrà rilasciata, questa invierà a Pico gli eventi insieme allo schema utilizzato per generarli.

Pico recupererà gli schemi dichiarati dagli eventi in arrivo per convalidarli, quindi salverà gli eventi ricevuti su BigQuery (figura 2.2).

I data analyst recupereranno gli eventi da BigQuery e, per interpretare le informazioni sull'evento, accederanno allo stesso web tool utilizzato dai product lead per definire gli schemi. Verranno visualizzati il nome, il tipo e la descrizione di ciascun campo dei diversi schemi di eventi; i data analyst accederanno agli schemi solo in modalità di visualizzazione (figura 2.3).

Il sistema generale è quindi suddivisibile in quattro elementi principali:

- **schema registry**: componente per la gestione (salvataggio, aggiornamento, lettura) degli eventi (informazioni che definiscono le azioni

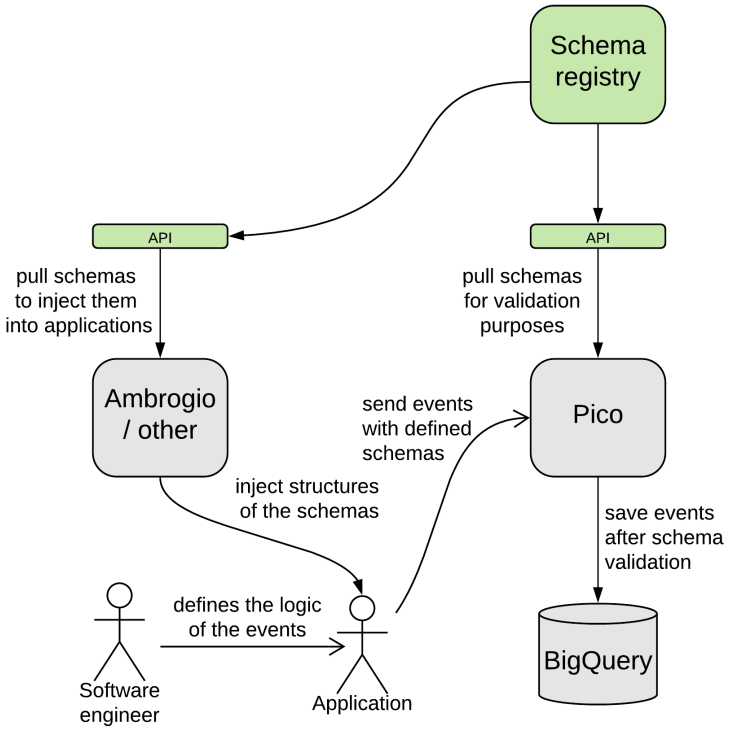

Figura 2.2: Diagramma che illustra le modalità di utilizzo del nuovo sistema da parte dei componenti del sistema pre-esistenti.

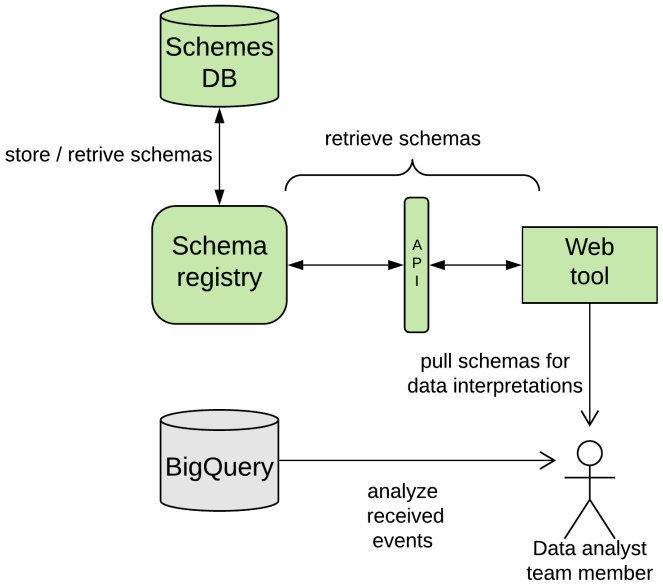

Figura 2.3: Diagramma che illustra le modalità di utilizzo del nuovo sistema da parte dei componenti data analyst, in fase di analisi degli eventi salvati.

tracciate) e degli schemi. Espone delle API per permettere agli altri componenti del sistema di interagire con gli schemi/informazioni;

- **web tool**: componente per permettere, attraverso un'interfaccia web che interagisce con lo schema registry, l'inserimento e la lettura di schemi di eventi delle varie applicazioni;
- **estensione per Ambrogio**: implementazione in Ambrogio della funzionalità che trasforma gli eventi di una data applicazione in codice Swift che verrà utilizzato nella fase di tracciamento degli eventi stessi;
- **estensione per Pico**: implementazione in Pico della funzionalità di verifica della struttura degli eventi utilizzando per la validazione lo schema utilizzato in fase di codifica dell'evento.

2.1.2 Formato dello schema

Come formato dello schema, è stato inizialmente deciso di adottare **JSON Schema**, a causa della sua universalità e facilità di comprensione anche a livello "umano". Il tipo di schema sarà reso automaticamente modificabile parametrizzandolo all'interno dell'identificazione dello schema stesso (sezione 2.1.6).

JSON Schema è una specifica per il formato basato su JSON per la definizione della struttura dei dati JSON; fornisce un contratto per la funzione di dati JSON all'interno di una determinata applicazione e come interagire con essi. I membri (o le proprietà) degli oggetti definiti dallo schema JSON sono chiamati *keywords* [4].

L'uso di questo tipo di schema comporta numerosi vantaggi:

- descrive rigorosamente la struttura e i vincoli di convalida dei dati JSON;
- convalida le informazioni per garantire la qualità dei dati inviati dai client;
- può essere utilizzato per fornire una documentazione chiara e leggibile dall'operatore umano;
- verifica le API JSON verificando la risposta JSON della richiesta rispetto a uno schema JSON.

Attraverso le notazioni fornite da JSON schema è possibile descrivere una struttura dati, dal numero e dal tipo dei campi fino a definire regole più precise per i valori consentiti.

Le principali parole chiave utilizzate per descrivere un oggetto JSON sono:

- **$id**: identificatore univoco per lo schema;
- **$schema**: usato per dichiarare che un frammento JSON è in realtà un pezzo di schema JSON. Dichiara inoltre quale versione dello standard di JSON schema su cui è stato scritto lo schema;
- **type**: specifica il tipo di dati di un elemento dello schema (è possibile specificare più tipi);
- **required**: un array di zero o più proprietà richieste;
- **properties**: coppie chiave-valore su un oggetto. L'elemento properties è un oggetto, in cui ogni chiave è il nome di una proprietà e ogni valore è uno schema JSON utilizzato per convalidare quella proprietà.

È anche possibile avere logiche di controllo più rigorose utilizzando i seguenti parametri:

- **default**: viene utilizzata per fornire un valore predefinito per un elemento;

ACTION KIND	ACTION INFO	DATA TYPE
connection_error	__dev_trigger	string
	authentication_type	string
	page_number	string
	request	string

Figura 2.4: Esempio di dettaglio di un elemento contenuto nel foglio di calcolo.

- **examples**: viene utilizzato per fornire una serie di esempi che convalidano l'elemento rispetto allo schema. Non viene utilizzato per la convalida, ma può aiutare a spiegare l'effetto e lo scopo dello schema ad un lettore;
- **pattern**: è una regular expression utilizzata per esprimere vincoli.

2.1.3 Modellazione dell'evento

Rispetto alle informazioni contenute nel foglio di calcolo, un evento è modellato solo da una piccola parte di essi.

Infatti un evento sarà composto da:

- un **action kind**
- 0-n istanze di **action info**

In figura 2.4 viene riportato un esempio della struttura di un possibile evento da rilevare.

A partire dalla struttura sopra definita possiamo costruire il JSON Schema mostrato nel listato 2.1.

```
{
   "$id":"https://example.com/event.schema.json",
   "$schema":"http://json-schema.org/draft-07/schema#",
   "title":"User event",
   "description":"Event management schema",
   "type":"object",
   "required":[
      "action_kind"
   ],
   "properties":{
```

```
      "action_kind":{
         "const":"connection_error"
      },
      "action_info":{
         "type":"object",
         "required":[
            "authentication_type",
            "page_number",
            "request",
         ],
         "properties":{
            "authentication_type":{
               "type": ["string", "null"]
            },
            "page_number":{
               "type": "number"
            },
            "request":{
               "type": ["string", "null"]
            }
         }
      }
   }
}
```

Listato 2.1: Esempio di schema che descrive le caratteristiche degli elementi di un evento.

Infine, un esempio di un evento conforme alla struttura modellata dal JSON schema riportato è rappresentato nel listato 2.2.

```
{
   "action_info":{
      "__dev_trigger": null,
      "authentication_type": null,
      "page_number": 14,
      "request": "https://example.com/"
   },
   "action_kind": "connection_error"
}
```

Listato 2.2: Esempio di evento conforme allo schema riportato nel listato 2.1.

2.1.4 Metadati degli eventi

Le informazioni rimanenti riportate nel foglio di calcolo che non si riferiscono direttamente all'evento stesso possono essere viste come "metadati" che descrivono i componenti dell'evento.

Anche se lo schema registry non li utilizza per la convalida degli eventi, sono comunque informazioni essenziali per la loro interpretazione da parte dei data analyst o in generale di coloro che dovranno consultarli; ai product lead verrà quindi chiesto di definire anche questi dati.

Le informazioni sono principalmente quelle elencate nella sezione 1.1.4.
Per quanto riguarda un' **action kind**:

- **action description**: descrizione dell'action kind
- **section**: raggruppamento concettuale di action kind per argomento (a scopo di leggibilità)

Per quanto riguarda un **action info**:

- **data type**: tipo di dato da tracciare;
- **description**: descrizione del parametro;
- **allowed values and examples**: valori consentiti, esempi vari;
- **data available from**: build number dell'applicazione dal quale tracciare l' *action info*;
- **required**: indica se il campo è obbligatorio o meno nella definizione dell'evento:
- **status**: indica lo stato dei diversi *action info*:
  - Not yet implemented
  - Implemented, not tested
  - Correctly implemented
  - Work in progress
  - Incorrectly implemented

2.1.5 Compatibilità dello schema

In linea generale, date due diverse versioni dello stesso schema (X-1 e X), le due versioni possono avere un diverso gradi di compatibilità [5]:

- **backward compatible**: producendo dati usando lo schema X-1, questi possono essere validati usando lo schema X;
- **forward compatible**: se si producono dati utilizzando lo schema X, è possibile convalidarli utilizzando lo schema X-1
- **full compatible**: se sono compatibili sia in modalità backward che forward.

Nel contesto analizzato, non esiste uno scenario in cui è necessario produrre dati con lo schema X e leggerli con lo schema X-1; pertanto, non è prevista la compatibilità forward.

In generale, l'idea è di adottare il seguente approccio:

- nella fase di evoluzione di uno schema esistente, l'utente può modificare lo schema stesso in modo da mantenere una compatibilità backward (esempio di modifiche: aggiunta di campi con valori predefiniti, rimozione di campi);
- non è possibile apportare modifiche allo schema che "interrompono" questa compatibilità in alcun modo (ad esempio, modificando il tipo di dati di un campo specifico);
- a causa della successiva analisi delle informazioni modellate utilizzando gli schemi sopra menzionati, è necessario prevenire modifiche tali da avere una situazione di riutilizzo di nomi già utilizzati in precedenza, anche con lo stesso tipo di dato (poiché ciò si rifletterebbe nell'avere, a livello di tabella BigQuery, una colonna con informazioni semanticamente diverse).

2.1.6 Versionamento degli schemi

Una caratteristica già menzionata nella sezione dei requisiti è il versioning degli schemi. Durante la sua esistenza, uno schema può evolversi, cambiando parzialmente la sua struttura (come discusso nella sezione precedente). Vogliamo mantenere un riferimento univoco a ciascuna versione dello stesso schema al fine di:

- accedere sempre alla versione dello schema con cui sono stati codificati i vari eventi, per poter effettuare i relativi confronti;
- avere conoscenza sull'aggiunta/rimozione di campi di un determinato schema, informazioni utili nell'analisi degli eventi salvati.

Sarebbe anche utile utilizzare un riferimento univoco alle varie versioni di uno schema per poterle definire/trovare senza errori. Un esempio di come potrebbe apparire un identificatore è: "bundle (app) _id / event_type / schema / versioning", dove:

- **bundle_id** (o **app_id**) è un riferimento all'applicazione alla quale è associato l'evento
- **event_type** è un riferimento all'evento che viene modellato con lo schema corrente
- **schema** indica il tipo di schema utilizzato dal sistema per definire l'evento (come indicato nella sezione 2.1.2) per astrarre la logica dal tipo di formato utilizzato
- **versioning** è l'effettivo riferimento alle diverse versioni di un determinato schema (vedi sotto)

Per la gestione delle versioni di uno schema (parametro versioning dell'ID), è possibile utilizzare la modalità *SchemaVer* [1], che presenta una logica numerica basata su tre parametri, **MODEL-REVISION-ADDITION** (es. 1-0-1).

- **MODEL**: viene incrementato quando viene apportata una modifica allo schema e questa modifica rompe la compatibilità e non consente l'interazione con i dati storici
- **REVISION**: viene incrementato quando viene apportata una modifica allo schema e questa modifica potrebbe portare alla non compatibilità con alcuni dei dati storici (a causa della possibilità che lo schema precedente fosse stato utilizzato per codificare le informazioni con alcune modifiche personalizzate)
- **ADDITION**: viene incrementato quando viene apportata una modifica allo schema e questa modifica lascia lo schema compatibile con tutti i dati storici.

Nel contesto in analisi, la possibilità di revision non esiste poiché nessuna entità esterna può modificare gli schemi.

2.2 Design di dettaglio

In questo capitolo vengono analizzate le scelte operate a livello di design nel dettaglio. In particolare viene analizzata la struttura è stata definita per ciascuno dei componenti, ed i principali pattern di progettazione adottati.

2.2.1 Schema registry

Lo schema registry è il componente principale del nuovo sistema. All'interno dello stesso infatti, fungerà da "motore" centrale che alimenterà poi le richieste degli altri componenti che interagiscono con lo stesso.

Questo elemento sarà quindi modellato come un servizio RESTful che esporrà delle API per permettere le operazioni sugli eventi e sugli schemi, in linea con quanto necessario per il funzionamento dell'intero sistema (dinamiche espresse nel capitolo 1).

Possiamo definire a livello progettuale due principali rami dell'architettura del sistema. Il primo si occupa dell'interazione con gli eventi, intesi come insieme di informazioni che descrivono una particolare azione che deve essere tracciata; il secondo invece tratta il rapporto con gli schemi veri e propri, contenenti le informazioni che definiscono la struttura dell'azione.

La gestione degli eventi e degli schemi presenta un'origine comune; una volta che lo schema registry riceve delle informazioni riguardanti un'azione da tracciare, queste vengono catalogate e salvate in due differenti contesti. I dati vengono infatti salvati in relazione ad un'entità di evento, ed allo stesso tempo vengono generati e separatamente salvati una serie di schemi (uno per ogni tipologia di serializzazione supportata) sempre relativi all'evento in questione. Lo stesso avviene quando viene aggiornato un particolare evento (in base alle dinamiche riportate nella sezione 2.2.1.2).

Una volta salvati, gli schemi sono delle entità separate dagli eventi che li hanno generati. È possibile a questo punto considerare quindi gli schemi come una entità in "sola lettura"; non vi è infatti diretto controllo sulla loro creazione, ma è possibile richiederli al sistema utilizzando diversi parametri (è possibile ottenere uno schema solo in uno dei formati di serializzazione supportati).

2.2.1.1 Servizi

Per soddisfare le richieste sulla base del comportamento appena descritto, lo schema registry esporrà le proprie funzionalità attraverso alcuni endpoints per gli eventi definite nella tabella 2.1, che permetteranno ai componenti che comunicheranno con lo schema registry di:

Ref.	HTTP verb	Path	Responses
1	POST	/events/{bundle_id}	201, 400, 409
2	GET	/events/{bundle_id}	200, 400, 404
3	PATCH	/events/{bundle_id}/{event}	200, 400, 404
4	GET	/events/{bundle_id}/{event}	200, 400, 404
5	GET	/events/{bundle_id}/{event}/{version}	200, 400, 404

Tabella 2.1: Tabella che riporta i servizi che dovranno essere esposti dallo schema registry per gestire l'interazione con gli eventi.

1. aggiungere un nuovo evento: inserendo nel request body le informazioni relative all'evento (riportate nel listato 2.3) queste vengono processate e in caso di esito positivo viene aggiunto all'insieme degli eventi, vengono inoltre creati e salvati i relativi schemi;

2. leggere tutti gli eventi di una data applicazione: tramite questo servizio è possibile richiedere al sistema le informazioni relativi agli eventi dell'applicazione passata come parametro;

3. modificare un evento esistente: passando tramite il request body una corretta istanza di evento è possibile modificare l'evento descritto dai parametri della route (se tale evento esiste)

4. leggere le informazioni di uno specifico evento: questa route permette di ricevere le informazioni riguardanti uno specifico evento, nell'ultima sua versione nota (quindi a fronte dell'ultimo aggiornamento effettuato);

5. leggere le informazioni di uno specifico evento ad una data versione: con questa chiamata è possibile recuperare le informazioni di uno specifico evento in relazione ad una precisa istanza di aggiornamento (indicata dal parametro "version").

```
{
  "bundle_id": "bundle_id",
  "event": "event name",
  "description": "event description",
  "section": "event section",
  "min_build": 0,
  "max_build": 100,
```

```
  "fields": [
    {
      "field": "field name",
      "data_type": "string",
      "description": "field description",
      "required": true,
      "allowed_values_examples": "field allowed values and examples",
      "min_build": 0,
      "max_build": 100
    }
  ]
}
```

Listato 2.3: Informazioni che compongono un elemento che modella un evento nel sistema.

Per quanto riguarda le routes per la gestione degli schemi (tabella 2.2) è possibile:

1. leggere tutti gli schemi: in base al formato richiesto passato come parametro è possibile recuperare tutti gli schemi salvati nel sistema;

2. leggere tutti gli schemi relativi ad un'applicazione: quando si utilizza questa route va specificato l'identificativo dell'applicazione di interesse per leggere tutti gli schemi relativi;

3. leggere lo schema di uno specifico evento: indicando il parametro "event" è possibile ottenere un singolo schema relativo ad un preciso evento;

4. leggere lo schema di uno specifico evento ad una data versione: anche per gli schemi è possibile ottenere una configurazione precedente di un determinato evento, indicando l'indicatore di versione.

2.2.1.2 Gestione di un evento

La gestione delle informazioni riguardanti un evento da parte dello schema registry è un aspetto chiave dell'intero componente.

Durante il suo ciclo di vita un evento interagisce con il sistema:

- quando viene inserito per la prima volta

- quando viene successivamente modificato.

Ref.	HTTP verb	Path	Responses
1	GET	/schemas/{schema_format}	200
2	POST	/schemas/{schema_format}/ {bundle_id}	201, 400, 409
3	GET	/schemas/{schema_format}/ {bundle_id}/{event}	200, 400, 404
4	PATCH	/schemas/{schema_format}/ {bundle_id}/{event}/{version}	200, 400, 404

Tabella 2.2: Tabella che riporta i servizi che dovranno essere esposti dallo schema registry per gestire l'interazione con gli schemi.

Nel primo caso vengono semplicemente eseguite le procedure di controllo riguardo la correttezza e completezza dei dati inseriti. In caso sia tutto corretto e l'evento che si tenta di inserire non sia già presente all'interno del sistema, questo viene aggiunto alla lista degli eventi, e i relativi schemi vengono creati.

La situazione è differente quando un evento esistente viene aggiornato. In questo caso infatti, l'azione intrapresa varia in base all'entità della modifica effettuata; in caso di update infatti (come descritto dal diagramma 2.5):

1. viene controllata la compatibilità con l'evento precedente: in base al set di regole impostate viene decretato se l'evento è compatibile con la versione precedente. Le regole possono essere modificate in base alla rigorosità che si vuole attuare per il mantenimento della compatibilità; per le esigenze espresse in fase di analisi attualmente viene controllato solamente che il tipo di dato dei vari field che compongono l'evento non sia mutato. In caso negativo di mancata compatibilità, la procedura termina;

2. viene verificato se è necessaria una nuova versione dell'evento: questo si verifica quando le modifiche effettuate non impattano solo su quelli che sono stati definiti in precedenza "metadati" dell'evento, ma che ne modificano direttamente le caratteristiche e/o la struttura. Come per la compatibilità, il tutto viene gestito come una sorta di set di regole, e quindi tutto in modo molto flessibile; allo stato attuale le modifiche che rendono necessaria una nuova versione sono: aggiunta di un field, rimozione di un field, modifica al livello di requirment di un field. Se si rende necessaria una nuova versione:

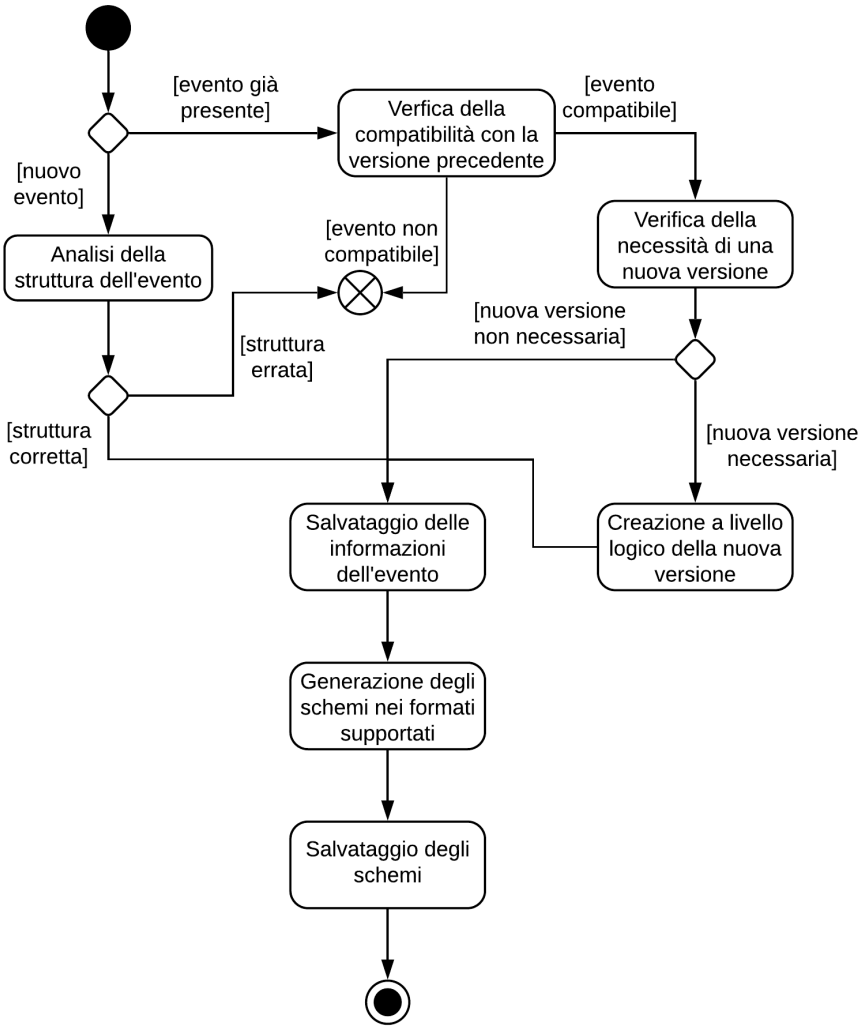

Figura 2.5: Diagramma che rappresenta in modo schematico le principali attività protratte dal sistema per la gestione dello schema di un evento.

- questa viene creata a livello logico;
- vengono salvate le informazioni relative alla nuova struttura dell'evento;
- vengono generati i relativi schemi in base alle tipologie di serializzazione supportate.

Se invece non fosse necessaria, vengono semplicemente aggiornate le informazioni relative all'evento.

2.2.1.3 Memorizzazione delle informazioni

Per la memorizzazione dei dati il servizio utilizza un database relazionale strutturato come mostra la figura 2.6:

- per la rappresentazione logica di un evento è stata definita una tabella per i dati relativi all'evento stesso e un collegamento con 0-n elementi salvati nella tabella fields, che rappresentano appunto i campi relativi all'evento in questione;
- gli schemi vengono salvati in una tabella separata;
- è stata creata una tabella per la gestione delle versioni, che mette in relazione un dato evento con una configurazione arbitraria di fields, in modo da poter sempre ricostruire la struttura di una specifica versione.

Per il salvataggio dei dati nel sistema è stato utilizzato il pattern architetturale DAO (Data Access Object); grazie a questa astrazione si e potuto separare completamente l'accesso al database dalla logica applicativa. I principali metodi (rappresentati nella figura 2.7) riguardano la gestione degli eventi e quella degli schemi.

2.2.2 Web tool

Il web tool è il componente che verrà utilizzato per accedere (in lettura e in scrittura, in base al ruolo dell'operatore) alle informazioni contenute ed elaborate dallo schema registry.

Si tratterà quindi di un applicativo web che utilizzerà le API fornite dallo schema registry stesso.

L'applicativo dovrà gestire:

- autenticazione: dovrà essere possibile autenticarsi con le proprie credenziali. In questo modo, in base al ruolo ricoperto all'interno dell'organizzazione, sarà possibile effettuare alcune operazioni piuttosto che altre;

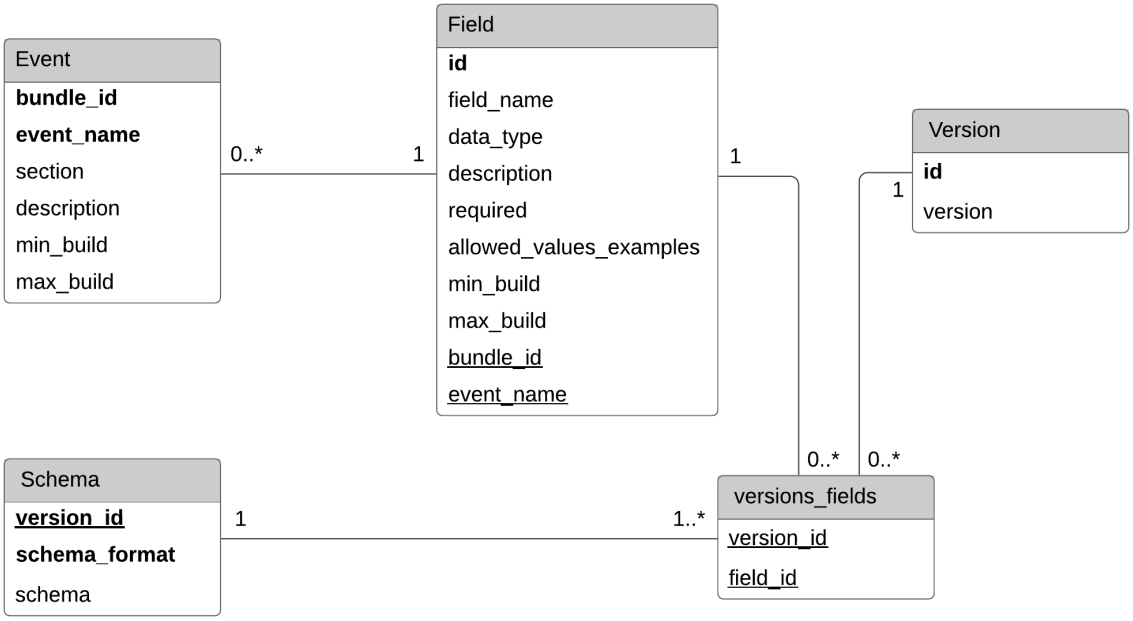

Figura 2.6: Diagramma che illustra la struttura base per il salvataggio delle informazioni generate dal sistema.

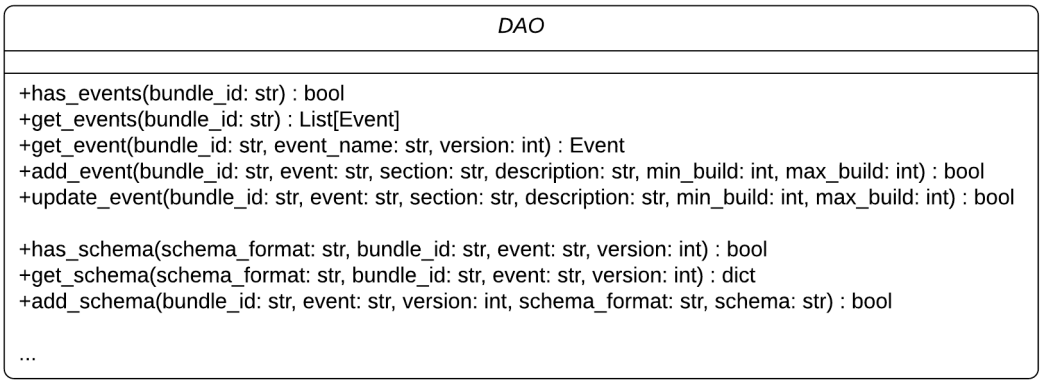

Figura 2.7: Elemento rappresentante le principali funzionalità del DAO per la gestione del database.

- visualizzazione degli eventi: l'utente avrà la possibilità di navigare gli eventi relativi ad una specifica applicazione, in modo da visualizzarne nel dettaglio i campi che lo compongono e le relative informazioni;
- aggiunta di eventi: l'utente autorizzato avrà la possibilità di aggiungere nuovi eventi relativi ad una data applicazione. In questo caso il sistema richiederà in input tutte le informazioni necessarie, dopodiché provvederà all'interazione con lo schema registry;
- modifica di eventi esistenti: l'utente autorizzato potrà modificare gli eventi presenti nel sistema, modificando i valori che è concesso di modificare.

In generale, dovranno essere mostrati eventuali errori ed indicazioni circa lo stato delle operazioni in corso.

2.2.2.1 Mockup

Attraverso l'utilizzo di un appositi servizio, Figma [6], sono state modellate le principali interfacce che comporranno l'applicazione. In questo modo sara piu facile in fase di implementazione creare un aspetto grafico che rispecchi i canoni definiti in fase di analisi e progettazione. Le schermate definite sono:

- home page con lista delle applicazioni (figura 2.8)
- schermata con lista degli eventi (figura 2.9)
- schermata di dettaglio di un evento (figura 2.10)
- schermata di aggiunta/modifica evento (figura 3.1)

2.2.3 Estensione per Ambrogio

Questo componente si occuperà della trasformazione delle informazioni di tutti gli schemi relativi ad una specifica applicazione in codice Swift, direttamente utilizzabile dall'applicazione.

Come già introdotto, l'operazione di creazione delle strutture dati atte al tracciamento delle azioni è attualmente svolta manualmente da un software developer che, utilizzando le informazioni contenute nel foglio di calcolo relativo, trascrive i nomi e i tipi dei dati. Questa operazione è altamente soggetta ad errori, che spesso vengono rimangono nascosti fino a fasi avanzate del progetto.

Grazie all'introduzione di un punto di conoscenza centrale quale è lo schema registry è ora possibile automatizzare in parte questo processo, in modo da

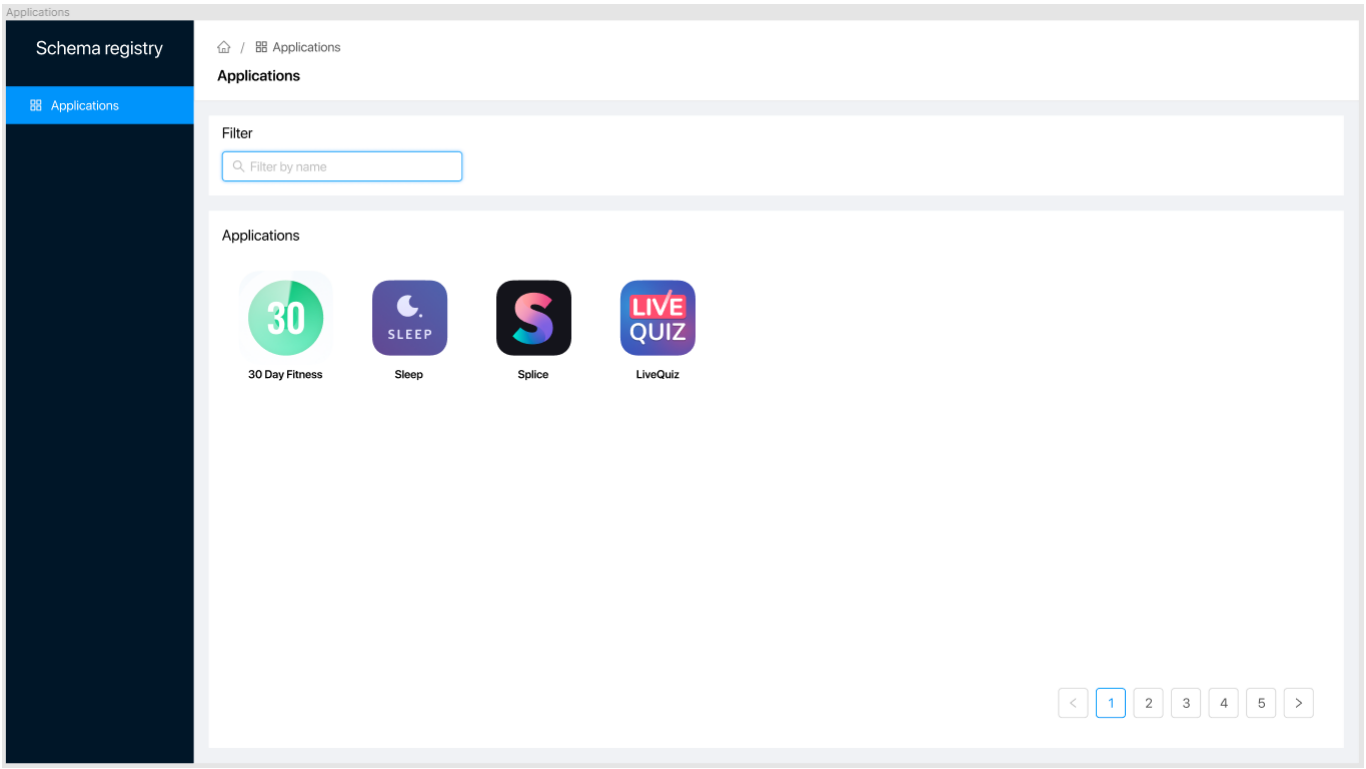

Figura 2.8: Mockup della schermata di presentazione delle applicazioni.

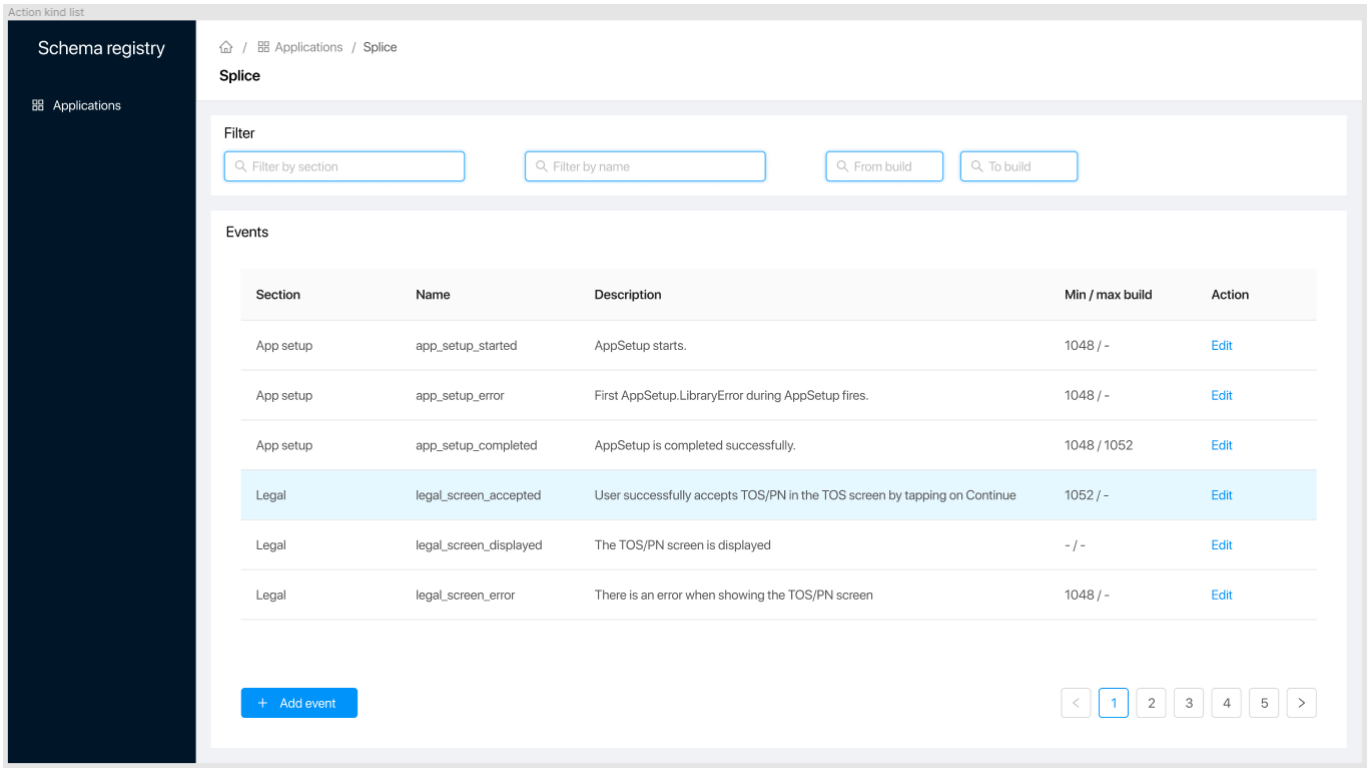

Figura 2.9: Mockup della schermata di presentazione degli eventi.

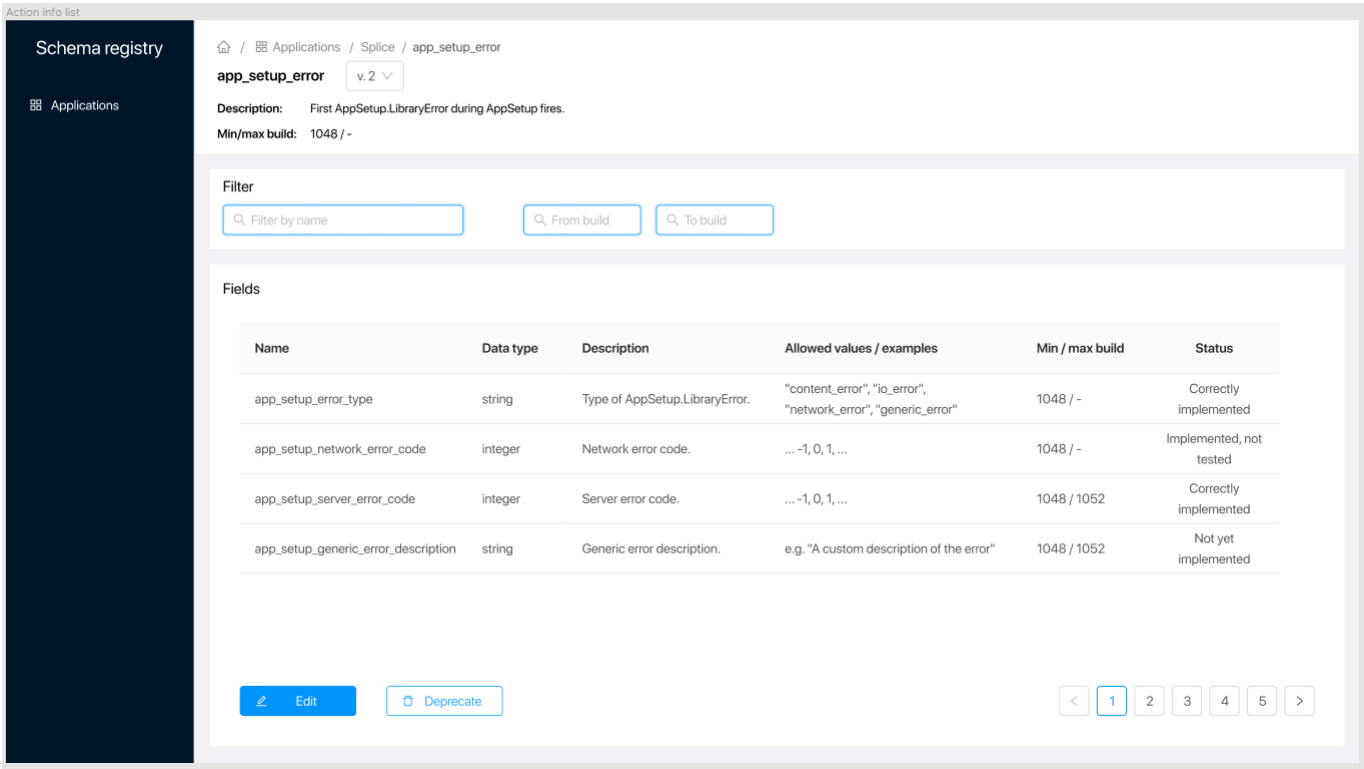

Figura 2.10: Mockup della schermata di dettaglio di un evento.

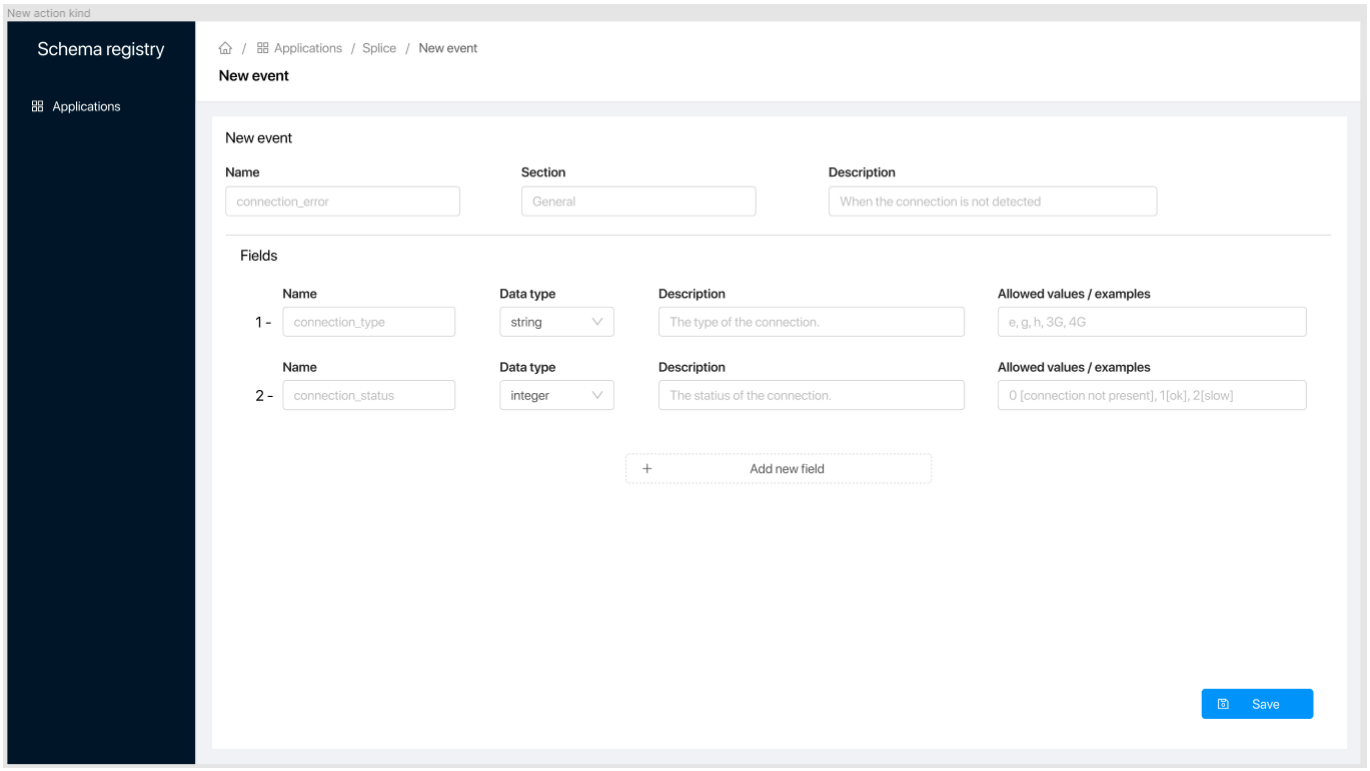

Figura 2.11: Mockup della schermata di aggiunta/modifica di un evento.

limitare il più possibile gli error. Questo modulo infatti dialoga con lo schema registry per recuperare tutti gli eventi attivi della applicazione in fase di sviluppo; una volta ottenuti procede a convertire le varie informazioni in codice Swift, in linea con la struttura definita (sezione 2.2.3.1) per far sì che questa sia integrabile con le attuali procedure già presenti nel sistema.

Una volta ottenuto il codice in Swift questo verrà integrato nelle routine che Ambrogio utilizza per iniettare il codice di preparazione all'interno delle applicazioni (sia in fase di primo sviluppo che in quelle di aggiornamento); in questo modo lo sviluppatore designato si ritroverà le classi relative alle azioni da tracciare, senza dover riportare nomi e avendo già i tipi dei singoli dati già impostati.

2.2.3.1 Struttura

La composizione del file Swift è stata definita in collaborazione con il team di software engineering in maniera tale che le nuove strutture riescano ad integrarsi correttamente con le procedure di Ambrogio.

Quindi:

- sono state definite delle strutture base, riportate nel listato 2.4, per permettere il giusto livello di astrazione e generalizzare così il tipo di dato che si sta elaborando;

- per ogni evento che presenta almeno un field si utilizza la struttura sopra descritta (un esempio di modellazione è riportato nel listato 2.5). Questa eredita da PicoData i due campi necessari per l'elaborazione lato server; questi due campi vengono quindi popolati con i dati relativi all'evento. In base al livello di requirement di ogni singolo field, viene scelto il tipo della variabile, in modo da rendere opzionali quei valori che non vanno raccolti obbligatoriamente. Questo modifica anche la logica di popolamento della mappa di valori, raccogliendo gli stessi solo se presenti.

- per ogni evento privo di field viene creata una struttura semplificata, che svolge la stessa funzione delle strutture presentate precedentemente ma elimina l'overhead di inizializzazione della mappa di valori che in questo caso non sono presenti. Queste strutture vengono raggruppate in un' *estensione* della classe `EmptyPicoData` (un esempio di modellazione è riportato nel listato 2.6).

```
protocol PicoData {
    var actionKind: String { get }
    var actionInfo: [String: Any] { get }
}

struct EmptyPicoData: PicoData {
    var actionKind: String

    var actionInfo: [String : Any] {
        return [:]
    }
}
```

Listato 2.4: Strutture di base che vengono utilizzate per generalizzare la logica del codice delle applicazioni.

```
struct MediaAdded: PicoData {
    let actionKind: String = "media_added"

   let projectId: String
   let numPhotos: Int?
   let numVideos: Int?
   let numVideoHighlights: Int?

    var actionInfo: [String : Any] {
        var data = [
            "project_id": self.projectId,
        ]

        if let numPhotos = self.numPhotos {
            data["num_photos"] = numPhotos
        }

        if let numVideos = self.numVideos {
            data["num_videos"] = numVideos
        }

        if let numVideoHighlights = self.numVideoHighlights {
            data["num_video_highlights"] = numVideoHighlights
        }

        return data
```

```
    }
}
```

Listato 2.5: Evento di esempio che traccia alcune *action info*.

```
extension EmptyPicoData {

    static var buttonPressed: PicoData {
        return EmptyPicoData(actionKind: "button_pressed")
    }

    static var buttonReleased: PicoData {
        return EmptyPicoData(actionKind: "button_released")
    }

    static var imageLoaded: PicoData {
        return EmptyPicoData(actionKind: "image_loaded")
    }
}
```

Listato 2.6: Estensionie per eventi di esempio che non presentano alcun *action info*.

2.2.4 Estensione per Pico

Questo componente sarà utilizzato, in fase di ricezione degli eventi, per validare le strutture degli stessi. Ogni evento infatti contiene un'indicazione circa il preciso schema che è stato utilizzato per la sua generazione; questo stesso schema dovrà quindi essere utilizzato per validarne la struttura.

Attualmente Pico (il componente del sistema che si occupa della gestione di tutti gli eventi generati dalle applicazioni) effettua una verifica sommaria basata su una serie di schemi di alto livello embeddati al suo interno. Attingendo dallo schema registry ora sarà invece possibile controllare più rigidamente gli eventi in entrata, utilizzando lo specifico schema di riferimento.

Per ogni evento quindi, il componente richiederà allo schema registry l'esatta versione dello schema che è stata utilizzata per la sua codifica, effettuerà i relativi controlli e valuterà la correttezza delle informazioni; la gestione di eventi incompleti o errati rimarrà invariata rispetto a quanto attuato dal sistema (salvataggio in una tabella generale, se possibile, altrimenti scarto dell'evento).

Capitolo 3

Implementazione

Quello che si andrà a descrivere in questo capitolo è la realizzazione dei componenti analizzati nella forma di proof of concept poiché non sono state prese in considerazione le dinamiche di integrazione e deployment all'interno del contesto aziendale corrente. Sono però da intendersi come componenti completamente definiti nelle logiche prese in considerazione, funzionanti e che possono interagire tra loro per portare a termine il lavoro per il quale il sistema è stato pensato e progettato.

3.1 Metodologie operative

Per descrivere correttamente l'attività svolta è possibile identificare tre fasi operative:

- **analisi della situazione attuale**: in questa fase e stata applicata la tecnica ad *intervista* per la raccolta delle informazioni necessarie alle fasi successive. Sono stati quindi organizzate diverse riunioni con le figure di riferimento dei team coinvolti nelle procedure in analisi, in modo da poter ricavare informazioni circa i punti chiave del sistema corrente e i principali problemi legati all'utilizzo e al mantenimento dello stesso;

- **progettazione delle nuove procedure e del nuovo sistema**: una volta identificati i principali problemi e criticità delle procedure attuali si e reso necessario rivedere l'intero sistema, prima in termini di ridefinizione dei processi passando poi alla progettazione del sistema in termini di caratteristiche tecniche delle componenti coinvolte. Durante questa fase si e operato in autonomia mantenendo una serie di riunioni cadenzate per aggiornare il responsabile dei progressi e per discutere delle criticità che mano a mano si manifestavano;

- **sviluppo dei componenti chiave del sistema**: una volta ricevuta l'approvazione del sistema progettato si e passati allo sviluppo dei suoi componenti chiave

Prima di questa fase è stata svolta una veloce introduzione alle principali tecnologie utilizzate per sviluppare e deployare i diversi componenti del sistema. Dopodiche si e operato in autonomia, utilizzando *git* [7] come strumento di versioning e appoggiandoci a *GitHub* [8] come repository remoto.

Il lavoro è stato mantenuto sotto controllo dal responsabile sottomettendo il codice attraverso il meccanismo delle *pull request*, in modo che questo venisse di volta in volta controllato ed approvato.

3.2 Schema registry

È il componente principale per la gestione degli eventi e dei relativi schemi. È composto quindi di un API che espone una serie di servizi con i quali gli altri componenti del sistema potranno interagire, al ne di poter utilizzare la conoscenza centralizzata sia in maniera automatica (da parte di tool come Pico e Ambrogio) che per scopi relativi all'interpretazione delle informazioni (da parte di operatori umani come i data analysts).

Questo componente è stato sviluppato utilizzando *Python 3.6* [9] in linea con le specifiche aziendali.

3.2.1 Struttura del componente

La struttura del progetto e stata organizzata in package, i quali contengono le classi che compongono il software; un package opera principalmente una suddivisione logica del codice, per una migliore organizzazione in fase di consultazione ed utilizzo dello stesso.

Il progetto è stato così suddiviso:

- **api_spec**: package che contiene un file yaml utilizzato per descrivere l'interfaccia esposta dal sistema (le sue routes e la natura dei parametri) utilizzando OpenAPI come specifica (in figura 3.1 la rappresentazione di una parte delle API definite);

- **instance**: contiene il riferimento all'oggetto che si occupa del mantenimento della persistenza nel sistema all'implementazione corrente;

- **schema_registry**: questo package contiene le classi che si occupano della logica del sistema per la gestione degli eventi e degli schemi da parte del componente. Presenta al suo interno:

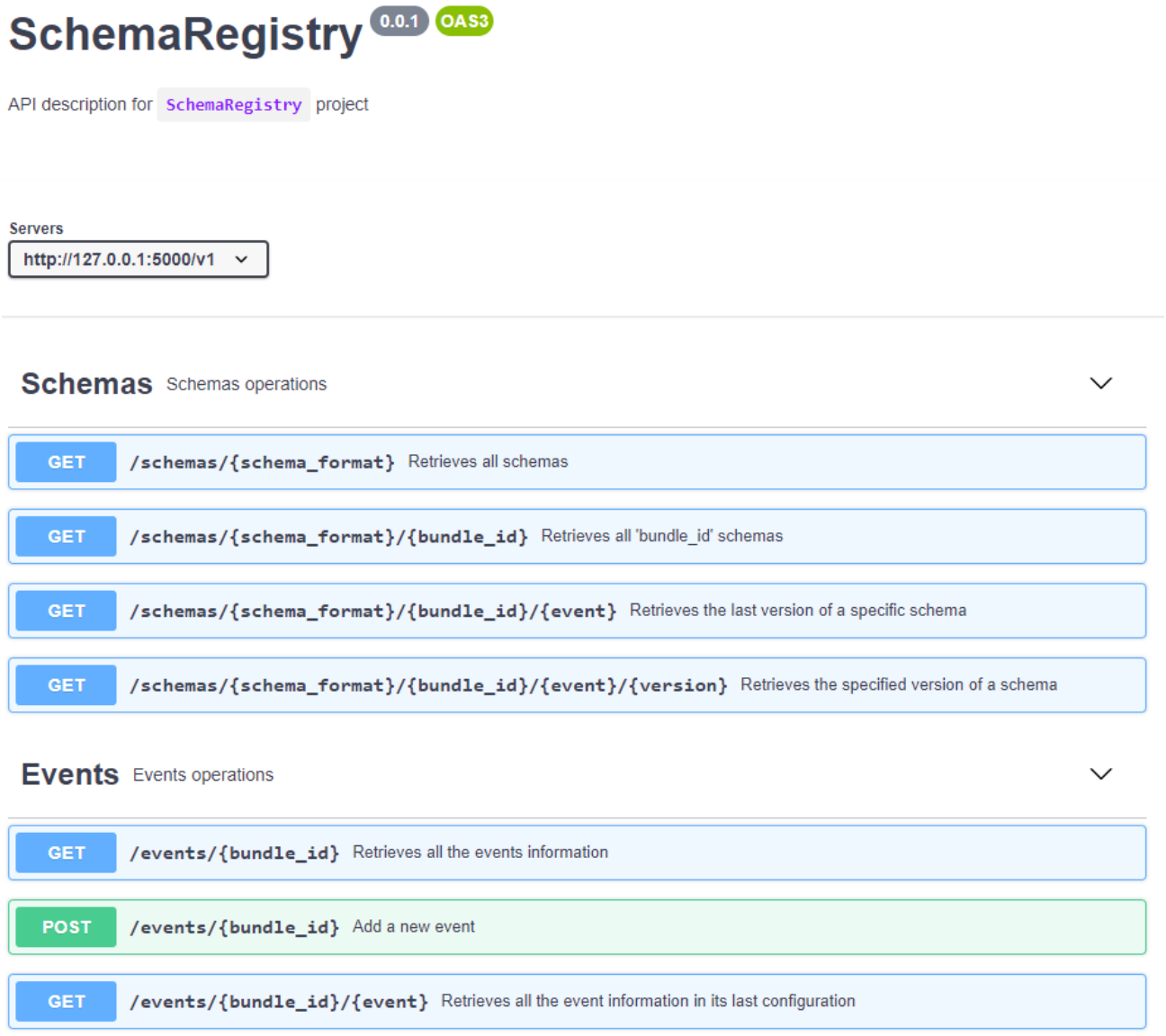

Figura 3.1: Rappresentazione delle API definite attraverso il tool Swagger [10].

  - **api**: package che raggruppa i controller del sistema stesso, ovvero classi che si occupano della logica vera e propria e della gestione delle informazioni passate allo schema registry attraverso gli endpoint esposti dal componente. La gestione degli endpoint stessi viene invece delegata a classi specializzate, contenute nel sotto-package denominato *routes*;
  - **models**: in questo package sono contenute le classi che implementano le strutture del componente e ne modellano la logica. Presenta inoltre le classi per l'utilizzo della persistenza in ottica di manager della persistenza stessa;
  - classi varie, che contengono principalmente metodi di utils e il componente di bootstrap del sistema stesso;

- **tests**: package che racchiude tutte le classi utilizzate per l'implementazione di test per il monitoring del codice prodotto.

3.2.2 Dettagli implementativi

3.2.2.1 Gestione di schemi ed eventi

Il compito del componente schema registry è rappresentato dalla gestione e memorizzazione strutturata di eventi e schemi.

Per quanto riguarda gli eventi, la loro gestione viene completamente delegata ai servizi esposti attraverso le API dello schema registry. Infatti attraverso queste ultime è possibile gestire completamente il "ciclo di vita" di un'entità evento, dalla sua creazione alla sua dismessa, passando per gli eventuali aggiornamenti; è quindi possibile recuperare i suddetti eventi in diverse modalità e per ognuno recuperare anche le singole versioni.

Diversa è invece la gestione degli schemi; questi vengono generati e salvati secondo precise specifiche e solamente in concomitanza dell'inserimento o dell'aggiornamento (se soddisfatti alcuni requisiti) di un evento. Attraverso i servizi messi a disposizione dallo schema registry sarà quindi possibile ai vari componenti del sistema il solo recupero degli schemi già presenti all'interno del database.

A causa di queste differenze logiche nella considerazione di queste due entità che possono essere considerate legate ma strutturalmente differenti, è stato deciso di gestirle separatamente. Per ognuna di esse quindi sono state create due classi, una che si occupa della gestione delle route che regolano l'esposizione delle operazioni che sono concesse sulla data entità (`event_router` e `schema_router`) ed una che invece racchiude la logica operativa che viene applicata per la gestione delle suddette attività (`event_controller` e `schema_controller`).

Le classi di routing presentano svariate funzioni, ognuna delle quali gestisce un particolare servizio che viene esposto a componenti esterni; di questo servizio vengono quindi indicati il path, il verbo HTTP e i parametri richiesti, il tutto gestito attraverso un sistema di annotazioni messo a disposizione dalle librerie utilizzate per lo sviluppo di tali API (per approfondimenti sezione 3.2.3). Queste funzioni delegano poi alle relative classi di controllo (che si occupano della gestione della richiesta) la formulazione della risposta da restituire al chiamante; un esempio è rappresentato dal listato 3.1.

```
@bp.route('/<bundle_id>', methods=['GET'])
@bp.route('/<bundle_id>/<event>', methods=['GET'])
@bp.route('/<bundle_id>/<event>/<version>', methods=['GET'])
@use_kwargs({
    "bundle_id": fields.Str(location='view_args', required=True),
    "event": fields.Str(location='view_args', required=False,
        missing=None),
    "version": fields.Int(location='view_args', required=False,
        missing=None),
})
def events(bundle_id, event, version):
    if event is None:
        return
            utils.send_response(event_controller.get_all(bundle_id))
    return utils.send_response(event_controller.get(bundle_id,
        event, version))
```

Listato 3.1: Funzione di routing per la gestione del recupero di evento dallo schema registry.

Le classi di controller offrono quindi diverse funzioni per la gestione della relativa entità. La logica comune di queste funzioni presenta una prima fase di controllo della correttezza dei valori passati come argomenti, una fase di elaborazione o recupero delle informazioni richieste, e in base al risultato ottenuto una fase finale di invio di una risposta al chiamante. Questa risposta viene modellata attraverso un'istanza della classe `BaseResponse`, che racchiude al suo interno il codice della risposta, il messaggio relativo ed eventualmente i dati da ritornare al chiamante; un esempio è rappresentato dal listato 3.2.

```
def get(bundle_id: str, event: str, version: int) -> BaseResponse:
    """
    Retrieve a specific event, based on given parameters.

    :param bundle_id: the reference to the application
    :param event: the reference to the event
```

```
    :param version: the reference to the version

    :return: a Tuple with the status code and a BaseResponse
        instance containing the message and eventually
    the retrieved event
    """
    if version is None:
        version = get_last_version(bundle_id, event)

    # if there is an event with the given bundle_id and event in the
        db, that event is returned
    if has_event(bundle_id, event, version):
        return BaseResponse(200, 'Event successfully retrieved',
            get_event(bundle_id, event, version))
    else:
        return BaseResponse(404, 'Event with provided information is
            not present')
```

Listato 3.2: Funzione della classe controller degli eventi per recuperare un evento salvato nel sistema.

3.2.2.2 Persistenza

Per la gestione della persistenza, come indicato in fase di progettazione (sezione 2.2.1.3) è stato utilizzato un database relazionale SQLite.

Per questo è stato definito lo schema del database utilizzando uno specifico file (contenuto come già indicato nel package instance) che viene utilizzato per la creazione iniziale delle varie tabelle.

Ad un livello superiore opera quindi una classe che implementa il pattern architetturale DAO; è stata quindi implementata una classe chiamata `generic_dao` che mette quindi a disposizione una serie di metodi per interagire con tutte le entità che il database gestisce, dagli eventi agli schemi veri e propri. Utilizzare questa logica architetturale permette di modificare agevolmente all'evenienza la tipologia del database; la classe che funge da DAO infatti offre all'utilizzatore una sorta di interfaccia per le funzionalità necessarie al sistema, senza implicare nessuna implementazione particolare, e di conseguenza rimane flessibile ai cambiamenti più o meno radicali del sistema di memorizzazione dei dati.

L'accesso al database viene invece gestito dalla classe `db_manager`, che presenta diverse funzioni per ottenere il riferimento al database, per inizializzarlo e per chiuderlo. La classe sfrutta il pattern *singleton* [11], grazie al quale riesce a mantenere le informazioni per la durata della sessione (schema in figura 3.2).

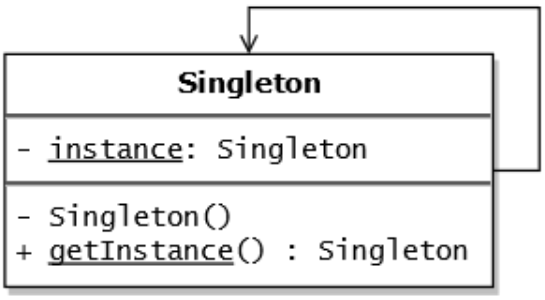

Figura 3.2: Diagramma UML esemplificativo di una classe che implementa il pattern *singleton*.

Il singleton infatti è un design pattern creazionale che ha come scopo quello di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

3.2.3 Tecnologie utilizzate

3.2.3.1 Librerie

Il core delle API comprendente l'esposizione dei servizi stessi è stato realizzato utilizzato **Flask**, un micro framework web (framework con poche o nessuna dipendenza verso librerie esterne) basato su Unicode e scritto in Python che offre strumenti, librerie e tecnologie che consentono di creare un servizio web [12].

Le sue principali caratteristiche sono:

- supporto integrato per unit test;
- gestisce richieste RESTful;
- comprende server e debugger per lo sviluppo;
- viene utilizzato Jinja2 per i template;
- supporta cookie di sicurezza (sessioni lato client);
- compatibilità con *Google App Engine*;
- estensioni disponibili per migliorare le caratteristiche desiderate.

Per facilitare lo sviluppo e offrire una maggior rigidità e sicurezza alle varie routes messe a disposizione dal componente ci si è serviti di webargs, una libreria Python per l'analisi e la convalida di richieste HTTP, con supporto integrato per i framework più popolari (tra cui appunto Flask) [13].

Utilizzando questi strumenti in maniera congiunta è stato possibile sviluppare in un tempo contenuto un sistema di API con precise specifiche per quanto riguarda la logica di gestione dei parametri e dei verbi HTTP, attraverso annotazioni e poche righe di codice.

Un'altra libreria che è stata utilizzata nello sviluppo, in questo caso per la gestione degli schemi veri e propri, è **jsonschema**, un'implementazione per Python della logica del formato JSON Schema. Questa libreria permette di interagire con schemi di tale formato in modo che sia possibile accedere direttamente a proprietà e valori, validare la struttura di uno schema ed iterare tutti i possibili errori [14].

3.2.3.2 Gestione delle dipendenze

Per la gestione delle dipendenze del progetto è stato utilizzato lo strumenti **pipenv**. La gestione delle dipendenze si verifica necessaria quando un componente B utilizzato nel sistema necessita di un componente A per funzionare; una situazione ancora più complessa si ha quando B necessita di A alla versione $x.y.z$,

Pipenv è uno strumento che ha come obiettivo riunire le migliori feature degli strumenti di gestione del mondo Python (bundler, composer, npm, cargo, yarn, ecc.). Crea e gestisce automaticamente un virtualenv per il progetto in questione, oltre ad aggiungere/rimuovere pacchetti dal *Pipfile* (listato 3.3) mentre vengono installati/disinstallati i relativi riferimenti. Genera anche un importante *Pipfile.lock* (listato 3.4) , che viene utilizzato per produrre build deterministiche [15].

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
webargs = "*"
flask-cors = "*"
jsonschema = "*"
pytest = "*"
coverage = "*"
```

Listato 3.3: Rappresentazione del file Pipfile utilizzato per la gestione dello *schema registry*.

```
{
    "_meta": {
        "hash": {
            "sha256":
                "2f1ec007d88b1021d6cabe529b689315abd4541748fc127599a8
             5437cee69a43"
        },
        "pipfile-spec": 6,
        "requires": {},
        "sources": [
            {
                "name": "pypi",
                "url": "https://pypi.org/simple",
                "verify_ssl": true
            }
        ]
    },
    "default": {
        "atomicwrites": {
            "hashes": [
                "sha256:03472c30eb2c5d1ba9227e4c2ca66ab8287fbfbbda388
            8aa93dc2e28fc6811b4",
                "sha256:75a9445bac02d8d058d5e1fe689654ba5a6556a1dfd8c
            e6ec55a0ed79866cfa6"
            ],
            "markers": "sys_platform == 'win32'",
            "version": "==1.3.0"
        },
       ...
    }
}
```

Listato 3.4: Porzione del file Pipfile.lock, che mostra come per ogni dipendenza tra le altre informazioni vengono specificati i riferimenti ad una determinata versione.

Grazie a questo strumento è possibile inoltre:

- effettuare il passaggio automatico tra la versione di sistema di Python con una versione definita dall'utente per il progetto;
- gestire automaticamente gli ambienti virtuali creati;
- sovrascrivere l'ambiente Python predefinito con uno dinamico;

- ricercare dei comandi delle varie versioni di Python.

3.3 Web tool

3.3.1 Struttura del componente

Anche questo componente è stato organizzato in package, per mantenere il codice strutturato e quindi più facile da modificare e mantenere.

La cartella di riferimento contiene i seguenti elementi:

- **Dependencies**: in questo package sono stati centralizzati tutti gli oggetti che vengono utilizzati, all'interno del progetto, però effettuare chiamate remote verso servizi esterni (principalmente per comunicare con lo *schema registry*). Queste funzioni vengono centralizzate per avere in un unico posto tutti i riferimenti necessari alle comunicazioni tra il componente e il "mondo esterno", in modo che sia più facile accedere e modificare le logiche relative;

- **Logic**: questo package contiene le classi utilizzate per controllare le logiche di recupero ed invio delle informazioni legate all'utilizzo del web tool. In queste classi vengono implementate le chiamate a servizi (utilizzando gli oggetti contenuti nel package *Dependencies*), delle quali sono gestite le risposte e i potenziali errori. Anche questa organizzazione è attuata per permettere una gestione modulare dei compiti all'interno del componente;

- **UI**: package che raggruppa tutte le classi utilizzate per modellare i componenti dell'interfaccia grafica del tool, nonché i riferimenti legati alla navigazione all'interno del sistema. Vi sono diversi sotto-package, in particolare:

  - **Applications**: in questo package sono raccolte le classi che gestiscono le schermate principali del web tool. Ogni classe racchiude anche la logica legata al funzionamento di quella particolare classe, per quanto riguarda l'interazione con l'utente e le relative risposte attese. Per effettuare operazioni che interagiscono con i dati del sistema vengono poi utilizzate funzioni contenute nelle classi racchiuse nel package *Logic*;
  - **Shared**: raggruppa gli elementi custom dell'interfaccia, che vengono utilizzati in più parti del codice e che quindi devono essere condivisi (form per la gestione dell'evento, raggruppamenti, loader, ecc.);

  - **Structure**: contiene la classe di riferimento che modella la struttura generale dell'applicativo web. Gestisce quindi le funzioni da eseguire al primo avvio, nonché la logica di navigazione determinata dalla tipologia di url che viene richiesto in un particolare momento dell'utilizzo;
  - classi varie, tra le quali `menuSection.ts` e `route.ts` che gestiscono le opzioni di navigazione presenti nel menù laterale e nella logica di impaginazione prevista;

- classi varie, che comprendono il componente di bootstrap del sistema ed elementi per la gestione dello stato. In particolare:
  - **Root.tsx**: rappresenta la gerarchia principale degli elementi che compongono il sistema. Vengono quindi riportate le strutture secondo una modalità a tag annidati, in modo da definire il contesto nel quale ogni elemento opera, organizzandosi in una logica *padre-figlio*;
  - **state.ts**: classe che mantiene i riferimenti ai principali classi di modello che vengono utilizzati all'interno del sistema;
  - **store.ts**: è l'elemento centrale che consentono ai componenti di inviare azioni e iscriversi agli aggiornamenti dei dati. Nell'uso tipico di un sistema sviluppato con react/redux, i componenti non devono mai preoccuparsi dei dettagli di gestione centralizzata, ma faranno mai riferimento direttamente allo store; inoltre quest'ultimo gestisce internamente i dettagli di come l'archivio e lo stato vengono propagati ai componenti collegati.

3.3.2 Tecnologie utilizzate

Per lo sviluppo di questo applicativo sono stati utilizzate in stretta relazione due librerie JavaScript, **React** e **Redux**; il tutto è stato sviluppato utilizzando come linguaggio **Typescript**.

3.3.2.1 React

React è una libreria JavaScript per la creazione di interfacce utente; è mantenuta da Facebook e da una comunità di sviluppatori. Questa libreria è stata pensata per la creazione di single-page applications o in generale applicazioni che girano sul browser, per sistemi guidati da dati che cambiano rapidamente e che necessitano di essere aggiornati. Oltre alla manipolazione dei dati le applicazioni React hanno una grande flessibilità grazie a delle librerie aggiuntive per gestire il routing, gestione degli stati e interazioni con API [16].

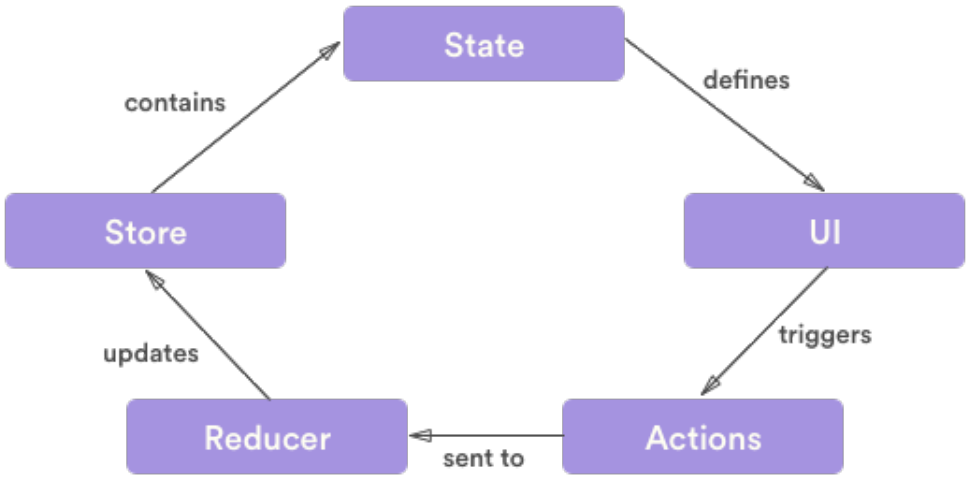

Figura 3.3: Rappresentazione del ciclo operativo che regola il funzionamento di un'applicazione che utilizza Redux per la gestione dello stato.

React utilizza un approccio dichiarativo e questo semplifica la creazione di applicazioni web, un'applicazione web viene strutturata in diversi **componenti**; in maniera approssimativa è possibile dire che un'applicazione realizzata con React è composta da tanti componenti quanti sono i vari elementi che compongono l'interfaccia. È poi compito dello sviluppatore decidere quali componenti creare e come strutturarli.

3.3.2.2 Redux

Redux è una libreria JavaScript open source per la gestione dello stato dell'applicazione; la logica che la governa è molto influenzata dalla programmazione funzionale, ed è principalmente utilizzata in stretta relazione con librerie per interfaccia come appunto React [17].

Il suo scopo è essenzialmente quello di rendere prevedibile il cambio di stato tramite l'imposizione di alcune restrizioni su come e quando questo può avvenire. Si basa sul concetto di avere un unico stato, rappresentato da un oggetto JSON e conservato in uno **store**. Questo stato può mutare solo in seguito ad azioni, ma non direttamente; la modifica avviene tramite l'invocazione di una funzione pura denominata reducer (come rappresentato in figura 3.3).

Redux si basa su tre principi:

- **single source of truth**: lo stato è l'unica fonte di verità a cui fa riferimento l'interfaccia utente;

- **state is read-only**: lo stato è in sola lettura e può mutare solo con un intento ben definito, a partire da una azione e attraverso un reducer;
- **changes are made with pure functions**: i cambiamenti allo stato avvengono solo attraverso *funzioni pure*, cioè una funzione che dato un certo input restituisce sempre lo stesso output senza *effetti collaterali*. Queste rende le funzioni facilmente prevedibili e testabili. Le pure function sono utilizzate per gestire i reducer.

Questi tre principi, per quanto possano sembrare limitanti, servono a garantire una corretto rispetto dei principi di coesione e accoppiamento nel software.

Capitolo 4

Valutazione

4.1 Testing

Durante lo sviluppo dei vari componenti sono state implementate delle procedure di testing del codice prodotto. Questo è avvenuto in particolare per il componente *schema registry* in quanto si tratta dell'elemento del sistema che più presenta una difficoltà nella logica che lo regola così come si presta, vista la sua natura, a procedure di testing automatico.

I moduli di test contengono test automatici per verificare la conformità delle classi dei moduli al comportamento previsto e intercettare possibili regressioni future quando il codice verrà modificato. Nella code-base di test che è stata sviluppata sono stati previste due principali tipologie di test:

- **unit test**: si tratta di test il cui scopo principale è controllare singole funzionalità di una classe (ovvero singoli metodi o persino singole righe di codice). Richiedono una conoscenza dettagliata della progettazione e del codice del programma interno, quindi sono scritti dai programmatori durante l'implementazione della classe in questione;

- **system test**: sono test che verificano il comportamento del sistema nella sua interezza, simulando una modalità di utilizzo da parte dell'utente finale/componente. È un tipo di test *black-box*, in cui il risultato viene controllato per verificare la correttezza del sistema; basano le definizioni sulle specifiche dei requisiti.

Sono state quindi definite diverse classi che si occupano della gestione dei diversi livelli di testing descritti. In particolare:

- **test_events** si occupa di gestire unit test per quanto riguarda la logica di gestione degli eventi (una delle due macro-aree che vengono gestite dalle API). In particolare si occupa di testare:

  - la validità dei campi di un evento, verificando che non vengano passati eventi al servizio dedicato che contengano campi duplicati, ovvero campi che presentano lo stesso nome;
  - la compatibilità tra eventi, occupandosi quindi di controllare che coppie di eventi siano o meno compatibili tra loro. È un aspetto fondamentale della gestione degli eventi stessi, quindi sono state prese in considerazione numerose casistiche, cercando di evidenziare i casi che devono essere evitati, come campi con lo stesso data type;
  - la necessità di creare una nuova versione dato un evento e la sua evoluzione. In questo set di test si verifica la logica per cui, se da una versione a quella successiva vengono rimossi o aggiunti campi o vi è una modifica nel livello di requirement di un campo allora il sistema deve segnalare la necessità di creazione di una versione differente da quella precedente.

- **test_schemas** racchiude i test utilizzati per controllare il meccanismo di creazione degli schemi a partire da eventi inseriti. Questo insieme di test si andrà quindi ad applicare a funzionalità interne del sistema (in quanto non vi sono servizi per la creazione degli schemi, ma questa avviene solo all'inserimento o all'aggiornamento di un evento), ma rimane comunque di fondamentale importanza per tenere sotto controllo quello che è uno dei processi fondamentali dello schema registry.

  Le funzioni di controllo principali verificano:

  - la creazione di uno schema a partire da uno specifico evento. Vengono quindi controllati i campi dello schema creato, poiché questi devono rispettare le dinamiche che mappano le informazioni dell'evento con quelle dello schema;
  - la gestione degli errori in caso di logiche errate all'interno dell'evento usato per generare lo schema. Sono casi che non devono verificarsi poiché a monte della gestione dell'evento vi sono comunque controlli che non permettono a quest'ultimo di essere in uno stato non consistente con le logiche definite, però si è deciso comunque di testare queste dinamiche in modo da capire, anche solo osservando i test relativi alla gestione dello schema, quali sono i comportamenti desiderati.

- la classe **test_app** si occupa infine di gestire un system test generale circa il funzionamento del sistema di API. In particolare viene quindi creato un contesto in cui si comunica con le suddette API attraverso una chiamata dei servizi esposti (simulando quindi l'interazione tra componenti esterni

ed API) e si verificano, dati alcuni scenari operativi, le relative risposte del sistema. Sono stati in particolare testati:

- inserimento corretto di eventi;
- inserimento di eventi che presentano errori strutturali (campi duplicati) o di eventi già presenti nel sistema;
- aggiornamento corretto di eventi;
- aggiornamento di eventi che presentano errori strutturali (campi duplicati, campi incompatibili) o di eventi non presenti nel sistema;
- recupero di eventi (tenendo anche in considerazione i casi in cui l'evento voluto non esiste o vengono utilizzati parametri non validi);
- recupero di schemi (tenendo anche in considerazione i casi in cui lo schema voluto non esiste o vengono utilizzati parametri non validi);

Per la gestione dei test è stato utilizzato pytest, framework di test che consente di scrivere codici di test utilizzando Python. Permettere di strutturare codice per testare qualsiasi cosa da database, API, interfaccia utente, ecc [18].

Le proprietà principali sono:

- sintassi semplice ed immediata che permette un utilizzo sistematico del framework;
- produce informazioni dettagliate circa le asserzioni non corrette;
- può eseguire test in parallelo;
- può eseguire uno specifico test o insieme di test;
- rileva automaticamente i moduli e le funzioni di test presenti nel progetto.

Grazie alle sue caratteristiche per strutturare un insieme di funzioni di test è sufficiente racchiudere le funzioni in una classe denominata *test_X* (dove X è essere una stringa qualsiasi), e denominare le funzioni *test_X* o *X_test* con la stessa logica; al comando di esecuzione dei test, il framework eseguirà automaticamente tutte le funzioni che rispettato quanto descritto (esempio: 4.1).

```
def test_new_version_needed(event, fields):
    # Same fields
    assert not new_version_needed(event, fields)

    # Add / remove fields
```

```
    assert new_version_needed(event, [])
    new_fields: List[Field] = fields[:-1]
    assert new_version_needed(event, new_fields)
    new_fields.append(Field.from_dict({
        "field": "another_field",
        "data_type": "string",
    }))
    assert new_version_needed(event, new_fields)

    # Change on requirement level
    fields_with_different_requirement: List[Field] = []
    for field in fields:
        fields_with_different_requirement.append(Field.from_dict({
            "field": field.field,
            "data_type": field.data_type,
            "required": field.required,
        }))

    assert not new_version_needed(event,
        fields_with_different_requirement)
    fields_with_different_requirement[0].required = not
        fields_with_different_requirement[0].required
    assert new_version_needed(event,
        fields_with_different_requirement)

    # No need of new version
    fields_with_different_metadata: List[Field] = fields.copy()
    fields_with_different_metadata[0].description = "Another
        description"
    fields_with_different_metadata[0].allowed_values_examples =
        "Example 1"
    assert not new_version_needed(event,
        fields_with_different_metadata)
```

Listato 4.1: Esempio di una funzione di test scritta in Python e compatibile con la logica del framework **pytest**.

Nel produrre codice di test è importante avere un'ordine di grandezza circa la capillarità dei test rispetto al codice principale; una metrica molto interessante in questo il **test coverage**, ovvero il grado di esecuzione del codice sorgente di un programma quando viene eseguita una particolare suite di test. Un programma con un'elevata coverage, misurato in percentuale, sta a signi-

Coverage report

Module ↓	statements	missing	excluded	coverage
schema_registry/api/event_controller.py	162	31	0	81%
schema_registry/api/routes/__init__.py	0	0	0	100%
schema_registry/api/routes/event_router.py	26	0	0	100%
schema_registry/api/routes/schema_router.py	19	0	0	100%
schema_registry/api/schema_controller.py	11	0	0	100%
schema_registry/app.py	22	1	0	95%
schema_registry/models/constants.py	4	0	0	100%
schema_registry/models/db_manager.py	21	0	0	100%
schema_registry/models/exceptions.py	6	1	0	83%
schema_registry/models/generic_dao.py	144	23	0	84%
schema_registry/models/objects.py	82	15	0	82%
schema_registry/utils.py	13	2	0	85%
Total	**510**	**73**	**0**	**86%**

Figura 4.1: Statistiche estratte attraverso il tool **coverage.py** dal codice del componente *schema registry*.

ficare che molto del suo codice sorgente è stato eseguito durante il test, il che suggerisce che ha una minore possibilità di contenere bug non rilevati rispetto a un programma con una coverage bassa.

Per gestire questo aspetto è stato utilizzato **coverage.py**, un tool per misurare la coverage dei programmi Python. Questo tool monitora il programma, rilevando quali parti del codice sono state eseguite, quindi analizza il sistema per identificare il codice che potrebbe essere stato eseguito [19].

In figura 4.1 viene riportato il risultato dell'esecuzione di tale tool sul progetto schema registry in questione; vediamo quindi le percentuali di copertura delle varie classi con vari altri parametri, e il risultato finale (86% di coverage).

4.2 Usabilità

Per quanto riguarda il web tool, il focus durante le fasi di progettazione dell'interfaccia e sviluppo dell'applicativo è stato maggiormente posto sull'usabilità del componente. Questo viene infatti utilizzato da operatori umani per consultare le informazioni contenute nello schema registry; in particolare, i data analyst hanno l'esigenza di utilizzare questo tool per un tempo prolungato e per un grande numero di analisi, il che comporta la necessità di facilità

ed immediatezza di utilizzo, per non creare overhead rispetto alla consultazione del foglio di calcolo. Al tempo stesso la logica che gestisce il componente è piuttosto semplice in quanto tratta la lettura e scrittura di eventi comunicando continuamente con lo schema registry.

Quindi, oltre alle indicazioni raccolte nella fase di analisi, sono state svolte attività di verifica dell'usabilità durante lo sviluppo e una volta terminato il prodotto. In particolare:

- durante tutta la fase di definizione del mockup sono stati consultati i principali team che andranno poi ad essere gli attori che avranno più a che fare con il tool. In questo modo si è cercato di focalizzare l'importanza della user experience attraverso fasi di raccolta di opinioni, dalla disposizione e scelta delle informazioni da mostrare alla posizione dei componenti all'interno delle schermate;

- una volta completato lo sviluppo del prototipo questo è stato presentato, assieme agli altri componenti del sistema, ad una serie di personalità interne all'azienda. Anche in questo caso, oltre a valutare il livello raggiunto in questi mesi di lavoro, è stata svolta una fase di raccolta di informazioni circa potenziali modifiche da apportare in termini di funzionalità e miglioramenti della user interface per ottenere un prodotto finale sempre migliore.

Conclusioni

L'obiettivo di questa tesi è stata la completa analisi di un sistema esistente e funzionante per la gestione delle metriche tracciate all'interno di applicazioni mobile, dalla quale partire per poi progettare un rinnovamento di questo sistema in termini di funzionalità e di procedure di attuazione delle medesime funzioni. È stato quindi richiesto di implementare i componenti principali di questo nuovo sistema, in modo che questi potessero interagire tra loro e portare a termine i compiti a loro designati, utilizzando nuove modalità più potenti e migliorate.

Per definire al meglio questo nuovo sistema si sono rese necessarie importanti fasi di studio della situazione attuale e dello stato dell'arte per le logiche di gestione degli schemi di dati, in modo da poter da una parte comprendere quali fossero i punti critici delle modalità fino ad oggi utilizzate all'interno del processo produttivo aziendale, e dall'altra riprogettare dapprima i processi operativi utilizzati dall'azienda per poi sviluppare i componenti del sistema al di sopra di questa nuova modalità.

Sono stati quindi introdotti i concetti di **evento** e **schema**, il primo per modellare le informazioni rilevanti circa un insieme di dati di interesse da tracciare nel contesto di utilizzo di un'applicazione e il secondo per descrivere secondo ben precise regole la struttura di tale evento, in modo che questa descrizione rigorosa possa essere utilizzata per attività di validazione delle informazioni scambiate all'interno del sistema.

L'insieme dei componenti sviluppati forniscono quindi funzionalità per la creazione, il salvataggio strutturato e la condivisione verso altri componenti di questo schemi ed eventi, relativi al parco applicazioni sviluppato dall'azienda. Queste informazioni, una volta inserite nel sistema, possono poi essere utilizzate da altri tool, consultati da operatori e gestiti nella loro evoluzione in termini di cambiamenti degli schemi stessi nel tempo.

Nella fasi di progettazione e implementazione del sistema e stata posta una particolare attenzione alla gestione dei diversi moduli, in modo da garantire agilità per fasi essenziali dello sviluppo di un software quali integrazione di nuove funzionalità e manutenzione di quelle esistenti.

Le principali problematiche che si sono presentate e che hanno poi influenzato svariate scelte, partendo dalle fasi di analisi fino all'implementazione dei componenti, riguardano l'altro numero di divisioni aziendali che andranno ad utilizzare questo sistema. Quanto sviluppato infatti sarà utilizzato in svariate fasi del processo produttivo, a partire dalla definizione delle metriche prima che un'applicazione venga sviluppata fino all'analisi dei dati inviati dalle applicazioni in produzione. Questo ha comportato il fatto che sono stati riscontrate svariate esigenze da parte dei diversi team, a volte anche potenzialmente in contrasto tra loro, e quindi alcune scelte progettuali sono state prese in ottica di avere il miglior trade-off tra quella che sarebbe la miglior scelta teorica e quelle che sono le esigenze espresse.

4.3 Sviluppi futuri

Il progetto sviluppato per questa tesi potrebbe essere migliorato ed esteso in diverse sue parti.

In particolare, per quanto riguarda il web tool, sarebbe utile una più profonda ed attenta progettazione dell'interfaccia e in generale un miglioramento della user experience, per garantire le massime prestazioni per coloro che utilizzano il componente.

Sarebbe poi utile implementare tutta una serie di funzionalità per snellire il più possibile le attività di definizione e consultazione degli eventi, evitando azioni ripetitive e quindi dispendiose in termini di tempo, come la possibilità di clonare la struttura di evento già presente nel sistema oppure la possibilità di una consultazione sommaria di tutte le caratteristiche principali di una serie di eventi appartenenti alla stessa applicazione.

Il componente schema registry potrebbe essere ottimizzato affinando alcune logiche di gestione degli schemi. Attualmente infatti, quando un evento viene salvato, viene generato il relativo schema nel formato standard (JSON Schema) ed è quindi possibile richiedere lo schema solo in tale formato. Una funzionalità interessante sarebbe prevedere una serie di altri formati supportati in modo che lo schema possa essere richiesto in differenti modalità, con la capacità del sistema di materializzare su richiesta lo schema nel formato desiderato.

A livello generale del sistema è necessario progettare ed implementare la gestione dell'autenticazione sia per le attività di inserimento e recupero delle informazioni dallo schema registry (in modo che possa essere differita in base al componente del sistema che richiede le informazioni) che nell'utilizzo del web tool. In quest'ultimo è necessario gestire anche il concetto di ruoli (come specificato in fase di analisi, sezione 1), in modo che in base al ruolo dell'utente che

utilizza il tool siano o meno abilitate le attività di modifica delle informazioni. Inoltre bisogna sviluppare l'estensione per Pico che consiste nella funzionalità di validazione degli eventi ricevuti dal sistema citato; le caratteristiche di tale estensione sono state prese in considerazione nella fase di analisi.

Sempre a livello generale, alla fine dei processi sarà necessario deployare/integrare tutti i componenti sviluppati all'interno delle specifiche aree aziendali.

4.4 Considerazioni finali

La realizzazione di questo progetto è stata un'esperienza profondamente stimolante e sfidante sotto numerosi aspetti, tra i quali troviamo sicuramente il doversi confrontare con una grande realtà aziendale con dinamiche operative complesse e strutturate.

L'analisi dei processi che coordinano un'attività importante come la gestione delle metriche delle applicazioni è stata molto interessante non solo per le ovvie implicazioni legate alla progettazione del nuovo sistema, ma anche per comprendere quali sono le sfide e i problemi che devono essere affrontati per gestire un tale mole di informazioni e utenze.

Il risultato ottenuto è quindi rappresentato da una serie di componenti completamente funzionanti in grado di applicare le logiche per le quali sono stati progettati e capaci di comunicare tra loro per gestire il ciclo completo dell'utilizzo delle metriche (partendo quindi dalla definizione delle stesse fino all'utilizzo terminale da parte di Ambrogio e dei data analyst).

Durante la fase di progettazione e sviluppo sono stati numerose le attività di confronto tra team ed utilizzatori del sistema, e questo ha permesso un confronto e uno scambio di opinioni ed idee fondamentali per la buona riuscita di un sistema di tale complessità ed importanza all'interno del contesto aziendale.

In conclusione il sistema creato svolge correttamente le funzioni per cui è stato progettato e per cui era stato in primo luogo pensato e voluto. Ovviamente i componenti, nella loro natura di *proof of concept* sono un punto di partenza per una fase di miglioramento ed integrazione nei sistemi attuali che li porteranno poi ad essere operativi ed utilizzati in prima linea. Personalmente mi ritengo soddisfatto del risultato finale e del percorso che è stato svolto per portare a termine quanto descritto in questa tesi.

Ringraziamenti

Desidero a questo punto ringraziare tutti coloro che sono stati al mio fianco e che mi hanno supportato in questo percorso di studi giunto ormai al termine.

Ringrazio innanzitutto il Professor Viroli che mi ha permesso di affrontare questo progetto di tirocinio e di tesi per me di così grande interesse, e per avermi professionalmente guidato e consigliato durante ogni fase di questo percorso. Un ringraziamento speciale anche alla Dott.ssa Ghironi e all'azienda Bending Spoons, che mi hanno calorosamente accolto ed offerto la possibilità di operare in un ambiente così professionale e stimolante.

Un ringraziamento quindi a tutti i miei amici, dentro e fuori l'università.

Un grazie immenso agli amici di una vita, sui quali ho sempre potuto contare e che non mi hanno mai lasciato solo, nonostante tutto. E allo stesso modo ringrazio quelli "nuovi", persone incontrate durante il mio cammino e che hanno deciso di rimanere al mio fianco, dimostrandosi preziosi alleati, ogni giorno sempre di più.

Ed infine un ringraziamento speciale va a tutta la mia famiglia, ed in particolare a mia sorella e ai miei genitori. Grazie per le possibilità che mi avete dato e per esserci sempre stati, pronti a sostenermi incondizionatamente nei momenti difficili, in prima linea per festeggiare ogni traguardo raggiunto. Senza di voi questo lungo viaggio non sarebbe stato possibile.

Grazie a tutti, di cuore.

Bibliografia

[1] Snowplow, *Introducing SchemaVer for semantic versioning of schemas*, https://snowplowanalytics.com/blog/2014/05/13/introducing-schemaver-for-semantic-versioning-of-schemas/, 2014.

[2] Wikipedia, *BigQuery*, https://it.wikipedia.org/wiki/BigQuery.

[3] Wikipedia, *Google Cloud Platform*, https://en.wikipedia.org/wiki/Google_Cloud_Platform.

[4] *JSON Schema*, https://json-schema.org/.

[5] Confluent, *Schema Evolution and Compatibility*, https://docs.confluent.io/current/schema-registry/avro.html.

[6] *Figma*, https://www.figma.com/.

[7] Wikipedia, *Git (software)*, https://it.wikipedia.org/wiki/Git_(software).

[8] Wikipedia, *GitHub*, https://it.wikipedia.org/wiki/GitHub.

[9] *Python*, https://www.python.org/.

[10] *Swagger*, https://swagger.io/.

[11] Wikipedia, *Singleton pattern*, https://en.wikipedia.org/wiki/Singleton_pattern.

[12] *Flask*, https://palletsprojects.com/p/flask/.

[13] *Webargs*,
https://webargs.readthedocs.io/en/latest/.

[14] *jsonschema*,
https://python-jsonschema.readthedocs.io/en/stable/.

[15] *Pipenv: Python Dev Workflow for Humans*,
https://pipenv-fork.readthedocs.io/en/latest/.

[16] *React*,
https://it.reactjs.org/.

[17] *Redux*,
https://redux.js.org/.

[18] *pytest: helps you write better programs*,
https://docs.pytest.org/en/latest/.

[19] *Coverage.py*,
https://coverage.readthedocs.io/en/coverage-5.0.3/.