Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Un servizio per l'amministrazione di una piattaforma "Software as a Service"

Tesi di Laurea Triennale

Relatore
Prof. Paolo Baldan

Laureando
Riccardo Michelon
Matricola 2042341

Anno Accademico 2023-2024

© Riccardo Michelon, Luglio 2024. Tutti i diritti riservati. Tesi di Laurea Triennale: "*Un servizio per l'amministrazione di una piattaforma "Software as a Service"*", Università degli Studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita".

"Imperare sibi maximum imperium est".

— Lucius Annaeus Seneca.

Ringraziamenti

Desidero esprimere la mia gratitudine verso il professor Paolo Baldan, mio relatore, per l'aiuto e il sostegno ricevuto durante la stesura dell'elaborato.

Per Anna, per la luce che sprigioni ogni giorno.
Per Elena, per l'esempio che sei sempre stata.
Per il nonno, per tutte le corse che hai fatto per renderci felici.
Per mamma e papà, per tutti gli insegnamenti e i sacrifici.

Per i valori affianco la quale sono cresciuto.

Padova, Luglio 2024 *Riccardo Michelon*

Sommario

Il presente documento illustra il lavoro svolto durante l'attività di stage presso *Datasoil S.r.l.*, della durata di trecento ore, focalizzato sullo sviluppo di un servizio completo, *backend* e *frontend*, per l'amministrazione della piattaforma proprietaria *SYN*. *SYN* è un prodotto che offre una soluzione completa di mappatura di *asset*, entità fisiche, in *digital twin*, loro rappresentazioni virtuali. La piattaforma consente agli utenti di gestire in tempo reale la posizione e le informazioni relative agli *asset*, fissi e mobili, anche interagendo con essi in realtà aumentata.

SYN è basato su un'architettura *Software as a Service multi-tenant*. Il servizio interno richiesto deve quindi permettere la creazione, la modifica e la supervisione dei diversi *tenant* legati ai vari clienti, garantendo l'isolamento e la correttezza delle informazioni. Lo strumento prodotto ha permesso di agevolare notevolmente la gestione e l'organizzazione delle soluzioni e delle configurazioni dei clienti, oltre che a offrire una nuova visione analitica dei dati attraverso l'implementazione di *dashboard*, consentendo all'azienda di condurre analisi statistiche e comportamentali volte a migliorare la qualità funzionale e commerciale del prodotto *SYN*.

Il lavoro ha compreso anche le attività di analisi dei requisiti e progettazione, mirate a trovare i punti di contatto con l'architettura esistente, facilitandone l'integrazione.

Indice

Acronimi e abbreviazioni xv

Glossario xvii

1 Introduzione 1
1.1 L'azienda . . . 1
1.2 Proposta di stage . . . 2
1.3 Descrizione del prodotto ottenuto . . . 3
1.4 Tecnologie e strumenti . . . 4
1.4.1 Tecnologie e strumenti organizzativi . . . 4
1.4.2 Tecnologie e strumenti per la gestione di progetto . . . 4
1.4.3 Tecnologie e strumenti di sviluppo . . . 4
1.5 Struttura dello stage e riepilogo delle attività svolte . . . 5
1.5.1 Pianificazione . . . 5
1.6 Struttura della relazione . . . 6
1.7 Convenzioni tipografiche . . . 6

2 Analisi dei requisiti 7
2.1 Analisi preliminare . . . 7
2.1.1 Obiettivi del prodotto . . . 7
2.1.2 Caratteristiche del prodotto . . . 7
2.1.3 Target di riferimento . . . 7
2.2 Lista dei requisiti . . . 8
2.2.1 Notazione . . . 8
2.2.2 Requisiti funzionali . . . 8
2.2.3 Requisiti di qualità . . . 11
2.2.4 Requisiti di vincolo . . . 11

3 Analisi tecnologica 13
3.1 Esplorazione tecnologica . . . 13
3.1.1 Module Bundler . . . 13
3.1.2 Libreria per l'implementazione delle dashboard . . . 14
3.1.3 Backend web framework . . . 14

4 Progettazione 19
4.1 Architettura infrastrutturale . . . 19
4.2 Architettura di sistema . . . 21
4.2.1 Modello architetturale . . . 21
4.2.2 Descrizione delle componenti . . . 22

4.2.3 Assemblaggio delle componenti . . . 22
4.3 Architettura delle componenti . . . 23
4.3.1 Frontend . . . 23
4.3.2 Backend . . . 25
4.3.3 Sistemi di persistenza . . . 35
4.3.3.1 Paradigma non relazionale . . . 35
4.3.3.2 Configurazione . . . 36
4.3.3.3 Master database . . . 36
4.3.3.4 Tenant Database . . . 36

5 Dettagli implementativi **39**
5.1 Autenticazione federata tramite JWT . . . 39
5.1.1 Processo di autenticazione . . . 39
5.1.2 AuthService, AuthContext e hook di supporto . . . 41
5.1.3 Protezione delle route . . . 44
5.1.4 Middleware backend di autenticazione . . . 44
5.2 Implementazione frontend . . . 45
5.2.1 Struttura organizzativa . . . 45
5.2.2 Routing . . . 45
5.2.3 Prime React e Prime Flex . . . 46
5.2.4 Implementazione form . . . 48
5.2.5 Implementazione grafici . . . 50
5.2.6 Interazione con le API del backend . . . 51
5.3 Implementazione backend . . . 52
5.3.1 Struttura organizzativa . . . 52
5.3.2 Utilizzo di Gin . . . 53
5.3.3 Procedura di binding . . . 54
5.3.4 Gestione degli errori . . . 54
5.4 Interazione con Mongo DB . . . 56
5.4.1 Middleware . . . 56
5.4.2 Tags e risultati delle operazioni di MongoDB . . . 56
5.4.3 Indici su MongoDB . . . 57
5.4.4 Transazioni . . . 58
5.4.5 Aggregation pipelines . . . 59
5.5 Documentazione . . . 61
5.5.1 Swagger . . . 61
5.5.2 Parametri descrittivi . . . 61
5.5.3 Processo di generazione . . . 61
5.5.4 Esposizione della documentazione . . . 62

6 Verifica e validazione **63**
6.1 Testing . . . 63
6.1.1 Test di unità . . . 63
6.1.2 Test di integrazione . . . 64
6.2 Validazione . . . 65

7 Integrazione e distribuzione continue **67**
7.1 Descrizione degli strumenti di CI/CD 67
7.1.1 Git e GitHub 67
7.1.2 Docker 68
7.1.3 AWS ECS e EC2 68
7.1.4 CircleCI 69
7.2 Ambienti di sviluppo, testing e produzione 70
7.3 Configurazione CI/CD tramite CircleCI 70
7.3.1 Jobs 72
7.3.2 Workflows 73

8 Conclusioni **75**
8.1 Consuntivo finale 75
8.2 Analisi critica del prodotto ottenuto 77
8.2.1 Stato di completamento attuale 77
8.3 Analisi retrospettiva e riflessioni finali 78

Bibliografia **i**

Elenco delle figure

1.1 Logo *Datasoil* 1
1.2 Logo *SYN* 1

4.1 Confronto tra *Non-SaaS* e *SaaS* 19
4.2 Schema dell'architettura *multi-tenant* 21
4.3 Architettura di sistema 22
4.4 Struttura dei sistemi di persistenza visti da *MongoDB Compass* 37

5.1 Interfaccia grafica della pagina di *login* 43
5.2 Interfaccia grafica del componente `DataTable` 48
5.3 Interfaccia grafica del form per la modifica di un *tenant* 49
5.4 Interfaccia grafica della *dashboard* delle *entities* 51
5.5 Interfaccia grafica della selezione degli indici 58
5.6 Schema di funzionamento dell'*aggregation pipeline* che calcola la progressione . 60
5.7 Interfaccia grafica del sito web della documentazione 62

7.1 Infrastruttura *ECS* con *EC2* e *Fargate* 69
7.2 Schema di funzionamento del *workflow CircleCI* 73

Elenco delle tabelle

1.1 Pianificazione delle attività. 5

2.1 Lista dei requisiti funzionali. 11
2.2 Lista dei requisiti di qualità. 11
2.3 Lista dei requisiti di vincolo. 12

4.1 Esiti possibili di Alive. 26
4.2 Esiti possibili di GetMasterTenants. 26
4.3 Esiti possibili di GetMasterTenant. 27
4.4 Esiti possibili di GetTenantIndexes. 28
4.5 Esiti possibili di NewMasterTenant. 28
4.6 Esiti possibili di NewEfmModule. 29
4.7 Esiti possibili di UpdateMasterTenant. 29
4.8 Esiti possibili di UpdateTenantIndexes. 30
4.9 Esiti possibili di GetTenantUsersStats. 31
4.10 Esiti possibili di GetTenantUsersCount. 32
4.11 Esiti possibili di GetTenantInspectionsCount. 32
4.12 Esiti possibili di GetTenantGeoContainersStats. 33
4.13 Esiti possibili di GetIndexes. 33
4.14 Esiti possibili di GetIndex. 34
4.15 Esiti possibili di ApplyIndexToAllTenants. 34
4.16 Esiti possibili di GetUploadSignedURL. 35

8.1 Pianificazione delle attività. 76
8.2 Consuntivo finale. 77

Elenco dei codici sorgenti

5.1 Codice della funzione `hostedUILogin`. 41
5.2 Codice della funzione fetchTokens. 42
5.3 Codice della componente AuthProvider. 43
5.4 Metodo per generare il *middleware* di autenticazione. 44
5.5 Routing frontend. 46
5.6 Esempio di utilizzo di un componente PrimeReact. 47
5.7 Esempio di implementazione di un form. 49
5.8 Esempio di implementazione di grafico a linea. 50
5.9 Metodo apiPost di useHttpApi. 51
5.10 Metodo `createMasterTenant` di `useTenantsApi`. 52
5.11 Implementazione di `GetMasterTenantController`. 53
5.12 Utilizzo dei *tag* in `MasterTenant`. 54
5.13 Gestione dei risultati multipli. 55
5.14 Middleware di MongoDB. 56
5.15 Esempio di utilizzo di una transazione. 59
5.16 Esempio di *stage* di una *aggregation pipeline*. 60
5.17 Script di generazione della documentazione. 62
5.18 Attivazione della *route* di Swagger UI. 62

6.1 Esempio di test di unità. 64
6.2 Esempio di test di integrazione. 65

7.1 Struttura del file di configurazione di CircleCI. 71
7.2 Esempio di definizione di un job. 72

Acronimi e abbreviazioni

API Application Program Interface. 15, *Glossario:* API

ASP Application Service Provider. 20

AWS Amazon Web Services. 4

B2B Business To Business. 1, *Glossario:* B2B

BSON Binary JSON. 35

CD Continuous Deployment. 4, *Glossario:* CD

CI Continuous Integration. 4, *Glossario:* CI

CPU Control Processing Unit. 69, *Glossario:* CPU

DBMS Database Management System. 5, *Glossario:* DBMS

EC2 Elastic Compute Cloud. 4

ECS Elastic Container Service. 4

HTTP HyperText Transfer Protocol. 16, *Glossario:* HTTP

HTTPS HyperText Transfer Protocol over Secure Socket Layer. 73, *Glossario:* HTTPS

IDE Integrated Development Environment. 70, *Glossario:* IDE

ITS Issue Tracking System. 4, *Glossario:* ITS

JSON JavaScript Object Notation. 16

JWK JSON Web Key. 44

JWT JSON Web Token. 39

NoSQL Not Only Structured Query Language. 35

OAuth2 Open Authorization 2. 39

OOP Object Oriented Programming. 35, *Glossario:* OOP

PKCE Proof Key for Code Exchange. 39

REST Representational State Transfer. 25, *Glossario:* REST

S3 Simple Storage Service. 4

SaaS Software as a Service. 2, *Glossario:* SaaS

SSH Secure Shell Protocol. 73, *Glossario:* SSH

VS Code Visual Studio Code. 4

XML eXtensible Markup Language. 16

YAML Yet Another Markup Language. 68

Glossario

API Interfaccia che consente a diverse applicazioni *software* di comunicare tra loro, definendo un insieme di protocolli e strumenti per la loro interazione. 15

Asset Qualsiasi risorsa tangibile di valore per un'organizzazione. 1

Autenticazione federata Metodo di autenticazione che consente agli utenti di accedere a più applicazioni o sistemi utilizzando le stesse credenziali, facilitando l'integrazione tra diversi domini di sicurezza. 3

B2B Relazione commerciale tra due o più aziende, caratterizzata dalla fornitura di prodotti o servizi destinati all'uso aziendale piuttosto che al consumo finale. 1

Backend Componente di un sistema *software* che gestisce la logica dell'applicazione, l'elaborazione dei dati e la comunicazione con il sistema di persistenza. Viene generalmente eseguita su un *server*. 5

Bearer token Tipo di *token* di sicurezza utilizzato per l'autenticazione e l'autorizzazione nelle applicazioni web, che conferisce al portatore del *token* l'accesso a risorse protette. 25

CD Pratica di sviluppo *software* che automatizza il rilascio continuo delle modifiche al codice sorgente in produzione, garantendo un flusso costante di aggiornamenti e miglioramenti e favorendo uno sviluppo incrementale. 4

CI Pratica di sviluppo *software* che prevede l'integrazione continua delle modifiche al codice sorgente in un repository condiviso, facilitando la rilevazione e la risoluzione tempestiva dei conflitti. 4

CI/CD Metodologia di sviluppo software che combina integrazione continua (CI) e distrubuzione continua (CD) per automatizzare e migliorare il processo di rilascio del software. 61

Cloud computing Modello di erogazione di servizi informatici che permette l'accesso *on-demand* a risorse computazionali, come *server*, *storage* e applicazioni, attraverso la rete Internet. 19

CPU Unità centrale di elaborazione di un computer, responsabile dell'esecuzione delle istruzioni dei programmi e del controllo delle operazioni del sistema. 69

Database Collezione organizzata di dati strutturati, generalmente memorizzata e accessibile da un sistema informatico, utilizzata per gestire e recuperare informazioni in modo efficiente. 2

DBMS Sistema *software* progettato per consentire la creazione, la manipolazione e l'interrogazione di una o più basi di dati in modo corretto ed efficiente. 5

Debugging Processo di identificazione, analisi e rimozione di errori o *bug* nel codice sorgente di un *software* per garantire il corretto funzionamento di un programma. 4

DevOps Insieme di pratiche e strumenti che combinano lo sviluppo *software* (Dev) e le operazioni *IT* (Ops) per migliorare la capacità di un'organizzazione di fornire applicazioni e servizi in modo rapido ed efficiente. 4

Digital twin Replica virtuale di un oggetto fisico o di un sistema, utilizzata per simulare, analizzare e ottimizzare le prestazioni reali attraverso la raccolta e l'elaborazione di dati. 1

Framework Struttura di supporto per lo sviluppo *software* che fornisce un insieme di strumenti, librerie e *best practices* per facilitare la creazione di applicazioni. 14

Frontend Parte di un'applicazione *software* che interagisce direttamente con l'utente fornendo un'interfaccia grafica che gestisce la presentazione dei dati e le modalità di interazione. 5

Geocontainer Entità di *SYN* che rappresenta una struttura geografica o spaziale, come un edificio, un'area esterna, una fabbrica, o un'area specifica all'interno di un impianto industriale. 3

Hosting Servizio che consente di memorizzare, gestire e pubblicare un sito web o un'applicazione su un *server* accessibile tramite internet. 4

HTTP Protocollo di comunicazione utilizzato per trasferire informazioni sul web, consentendo il recupero e la visualizzazione di pagine web e risorse multimediali. 16

HTTPS Versione sicura del protocollo *HTTP* che utilizza protocolli di sicurezza per crittografare i dati trasmessi tra il client e il server. 73

IDE Ambiente di sviluppo integrato che fornisce agli sviluppatori una *suite* di strumenti per scrivere, testare e debuggare il codice sorgente. 70

ITS Sistema di gestione dei problemi che consente alle organizzazioni di segnalare, monitorare e gestire i *bug* e le richieste di funzionalità all'interno di un progetto *software*. 4

Machine learning Branca dell'intelligenza artificiale che utilizza algoritmi e modelli statistici per permettere ai computer di apprendere dai dati e migliorare le loro prestazioni nel tempo senza essere esplicitamente programmati. 78

Multi-tenant Architettura *software* in cui un'unica istanza di un'applicazione serve più clienti o organizzazioni, rappresentati da un *tenant*, mantenendo separati i loro dati. 2

OOP Paradigma di programmazione basato sul concetto di "oggetti", che possono contenere dati sotto forma di campi, spesso noti come attributi, e codice sotto forma di procedure, spesso note come metodi. 35

Open source Modello di sviluppo *software* in cui il codice sorgente è reso disponibile pubblicamente, permettendo a chiunque di visualizzarlo, modificarlo e distribuirlo. 4

Query Richiesta di informazioni o dati effettuata a un *database*, formulata utilizzando un linguaggio di interrogazione specifico. 35

Repository Archivio centralizzato utilizzato per memorizzare, gestire e tenere traccia delle versioni del codice sorgente e delle risorse di un progetto *software*. 4

REST Stile architetturale impiegato per la progettazione di API web per sistemi distribuiti che utilizza un insieme di principi e vincoli per definire modalità e vincoli di interazione tra le componenti di un sistema. 25

Runtime Fase di esecuzione di un programma *software*, durante la quale il codice sorgente o il compilato derivante da esso vengono eseguiti dal processore. 14

SaaS Modello di distribuzione del *software* in cui le applicazioni sono ospitate su un server remoto e fornite agli utenti tramite la rete Internet, generalmente su base di abbonamento. 2

SSH Protocollo di rete che consente una connessione sicura e crittografata tra due dispositivi. Viene utilizzato principalmente per l'amministrazione remota dei sistemi. 73

Tenant Cliente o organizzazione che utilizza una specifica istanza di un'applicazione *multi-tenant*, con dati e configurazioni isolati dagli altri *tenant*. 2

Timesharing Tecnica di condivisione delle risorse computazionali tra più utenti contemporaneamente, consentendo l'accesso a un sistema centrale attraverso l'allocazione di intervalli di tempo per ciascun utente. 20

Capitolo 1

Introduzione

1.1 L'azienda

Datasoil S.r.l. è una *startup* Business To Business (B2B)$_G$ nata nel 2016 che opera nel settore dell'industria 4.0. È situata a Padova ed è composta da 5 dipendenti.

Figura 1.1: Logo *Datasoil*

Il prodotto principale dell'azienda è *SYN*, una piattaforma che offre una soluzione completa di mappatura di asset$_G$, entità fisiche, in digital twin$_G$, loro rappresentazioni virtuali.

Il prodotto è rivolto principalmente ad aziende industriali e manifatturiere, con l'obiettivo di manipolare ed elaborare dati di natura caotica, ordinandoli e presentandoli all'utente tramite un'interfaccia organizzata e intuitiva. *SYN* consente di gestire in tempo reale le informazioni relative agli *asset*, localizzandoli su una mappa tridimensionale e permettendo agli utenti di creare *ticket* strutturati per avviare e controllare ogni fase degli interventi associati all'*asset* in questione. Data la sua capacità di acquisire dati da diverse fonti in *streaming*, *SYN* offre un'analisi accurata degli eventi, consentendo agli utenti di ottenere *insight* immediati e reagire prontamente.

Attraverso un'app mobile che sfrutta la localizzazione tramite *beacon* e *GPS*, gli operatori possono inoltre individuare rapidamente gli *asset* circostanti e interagire con essi in realtà aumentata.

Figura 1.2: Logo *SYN*

1.2 Proposta di stage

La proposta di stage nasce dall'esigenza dell'azienda di avere uno strumento interno in grado di fornire una visione d'insieme dei clienti della piattaforma *SYN*, permettendo inoltre la creazione e la modifica delle loro configurazioni.

Per poter capire a fondo la proposta, è necessario introdurre a grandi linee l'architettura ad alto livello di *SYN*, che verrà poi ampiamente discussa in (§ 4, Progettazione).

Il prodotto *SYN* è basato su un'architettura Software as a Service (SaaS)$_G$ multi-tenant$_G$.

A differenza del paradigma *on-premise*, utilizzato in maniera universale fino agli inizi degli anni 2000, quando il *software* veniva installato ed eseguito direttamente sulle macchine degli utenti interessati, tramite il modello *SaaS* è possibile distribuire un prodotto nel *cloud*, consentendo l'accesso da parte dei clienti tramite la rete *Internet*.

L'architettura *SaaS* trova particolari vantaggi se associata al concetto di *multitenancy*, per la quale una singola istanza *software* è eseguita da un server ed è fruita da diverse organizzazioni, ciascuna rappresentata da uno specifico tenant$_G$. Un *tenant*, in italiano "inquilino" o "co-proprietario", rappresenta un ambiente isolato, con le sue peculiarità specifiche, che condivide con altri *tenant* esclusivamente l'istanza *software* e le sue risorse. In un'architettura *multi-tenant*, il relativo strumento di amministrazione è progettato per partizionare virtualmente e dinamicamente i dati e le configurazioni di ogni *tenant*, in modo che ad ogni client risulti di interagire con un'istanza dedicata, isolata e personalizzata.

È possibile ricondurre il funzionamento di questa architettura analogamente a quanto accade nella realtà in un immobile complesso, composto da più unità abitative, che condividono le risorse di un unico stabile ma vedono il servizio erogato come a loro utilizzo esclusivo, con le proprie tariffe e un grado di personalizzazione circoscritto alla loro unità.

Dato l'utilizzo di questa architettura, è evidente come l'oggetto della proposta rappresenti per l'azienda uno strumento essenziale per l'amministrazione dei vari *tenant*. Senza di esso, la gestione delle configurazioni avverrebbe direttamente a livello di database$_G$, cioè modificando manualmente la configurazione dei *tenant*. Questo tipo di intervento non solo è soggetto ad errore umano, ma è molto dispendioso in termini di tempo e fatica, poiché presuppone che ogni operatore conosca la rappresentazione effettiva della configurazione e i possibili valori di ciascun parametro. Inoltre, la personalizzazione del prodotto *SYN* è molto profonda, poiché non riguarda esclusivamente questioni grafiche, ma include anche aspetti funzionali, legati ad esempio all'autenticazione e alla connessione con servizi esterni. Per questo motivo, è molto importante che queste operazioni vengano eseguite in un ambiente controllato, in grado di bloccare azioni dannose o contestualmente incoerenti.

Altre problematiche da gestire riguardano la sicurezza e l'isolamento dei dati dei diversi *tenant*. Data l'importanza dei dati trattati, l'applicazione da sviluppare deve essere protetta da un sistema di autenticazione robusto, che concede l'accesso alle funzionalità offerte solamente agli utenti selezionati, cioè ai responsabili e ai gestori dell'infrastruttura della piattaforma *SYN*.

L'applicazione dovrà inoltre essere in grado di gestire una grande quantità di dati. Per garantire ciò, è necessario utilizzare strutture dati e algoritmi opportuni, riducendo ove possibile la complessità delle operazioni e interagendo in modo efficiente con i vari *database*.

Oltre a questo, senza lo strumento in questione, è difficile avere una visione globale dei *tenant*. Nei dati contenuti all'interno dei *database*, è possibile estrarre informazioni utili ad analisi statistiche e commerciali. Per esempio, conoscendo la mole di un certo dato o la frequenza di utilizzo di una determinata funzionalità, è possibile confrontare più *tenant* e notare dei pattern comuni. Date queste osservazioni, l'azienda riuscirebbe ad avere una concezione

più precisa dei clienti e del modo in cui questi utilizzano il prodotto. Sfruttando i dati raccolti, l'azienda potrebbe capire i punti deboli e le mancanze del prodotto, alimentando il processo di miglioramento continuo, decidendo di implementare nuove funzionalità, ottimizzando ciò che è già presente e applicando nuove strategie di *marketing*.

1.3 Descrizione del prodotto ottenuto

Il prodotto *software* ottenuto permette all'azienda di avere una visione d'insieme dei loro clienti, consentendo operazioni di amministrazione e monitoraggio delle relative configurazioni e facilitando i processi di gestione di esse. È costituito da un'applicazione web formata da più pagine, navigabili attraverso un'apposita *sidebar*.

La piattaforma è protetta da un sistema di autenticazione federata$_G$, che garantisce l'accesso alle risorse e alle funzionalità dell'applicazione solamente ad un gruppo ristretto di dipendenti autorizzati dell'azienda. Per poter accedere all'applicazione è necessario inserire le credenziali dell'account personale del *Google Workspace* dell'azienda.

Una volta effettuato l'accesso, la pagina iniziale permette di visualizzare la lista dei *tenant*, mostrando per ciascuno le informazioni principali e rendendo disponibili funzionalità di ricerca, ordinamento e filtraggio per ogni attributo. Tramite questa lista, selezionando un *tenant* è possibile accedere ad una vista più dettagliata, che mostra ogni parametro della relativa configurazione e un insieme di *counter* che indicano il numero di utenti, *asset* e *ticket* del *tenant* selezionato. Questa pagina è strutturata in modo da organizzare le informazioni per argomento, suddividendole in varie sezioni, accessibili attraverso un apposito menu.

Il sistema offre agli utenti autorizzati dell'azienda la possibilità di creare e modificare le configurazioni dei *tenant* tramite la compilazione di form dinamici, in grado di adattarsi in base ai valori inseriti dall'utente, variando i campi descrittivi del *tenant*, mostrando avvisi informativi e riportando errori in caso di fallimento dell'operazione richiesta.

Il processo di modifica consente di aggiornare i parametri di una specifica configurazione a partire da quelli attuali. La relativa vista organizza i vari campi in diverse sezioni. Tra queste, è presente la sezione di selezione degli indici, che permette di attivare o disattivare indici di ricerca all'interno del *database* del *tenant* in questione a partire da una lista di default. Tutte le operazioni rese disponibili all'utente restituiscono un *feedback* attraverso un avviso a comparsa che contiene le informazioni riguardo la richiesta e il relativo esito.

Tramite un'apposita pagina, è possibile visualizzare la lista degli indici, organizzandola e filtrandola in base alle loro informazioni più rilevanti. In modo simile a quanto accade per i *tenant*, è possibile analizzare la totalità delle informazioni relative ad ogni indice in lista attraverso una pagina dedicata. In questa pagina è possibile visualizzare le chiavi e le opzioni dell'indice in questione e, attraverso una sezione apposita, è possibile applicare l'indice a tutti i *tenant* memorizzati nel sistema.

Tramite l'implementazione di *dashboard* analitiche, il prodotto è inoltre in grado di mostrare agli utenti grafici informativi, volti a raffigurare l'andamento del prodotto *SYN*, permettendo all'azienda di condurre analisi riguardanti l'utilizzo del loro prodotto e il comportamento dei suoi utenti. In particolare, sono state create *dashboard* che mostrano l'andamento temporale di varie entità della piattaforma *SYN*, tra cui utenti, *asset*, *ticket*, ispezioni e geocontainer$_G$.

Per ogni grafico presente è possibile impostare il periodo di riferimento da analizzare, selezionando periodi preimpostati o inserendo date precise, e specificare la risoluzione temporale, tra cui ora, giorno, mese e anno. Oltre a questo, è possibile impostare alcuni selettori boolea-

ni che abilitano o disattivano il calcolo cumulato e la procedura di *padding* del grafico, cioè l'inserimento di valori nulli in corrispondenza dei punti privi di dati.

Interagendo con i grafici attraverso il cursore del *mouse*, è possibile analizzare ogni punto presente e visualizzare eventuali informazioni ad esso associate, riportate in un'apposita etichetta a comparsa.

1.4 Tecnologie e strumenti

Di seguito vengono riportate le tecnologie e gli strumenti utilizzati, divisi per ambito di applicazione. Da questi elenchi sono stati esclusi elementi secondari e omessi alcuni particolari, illustrati poi in (§ 3, Analisi tecnologica) o in altre sezioni ritenute più opportune.

1.4.1 Tecnologie e strumenti organizzativi

L'azienda si serve di strumenti noti e standardizzati per organizzare il lavoro e la comunicazione interna. Per facilitare la collaborazione da remoto, *Datasoil* utilizza *Slack* come piattaforma di messaggistica e *Google Meet* come strumento di videoconferenza per le riunioni interne. Per gestire la conoscenza e agevolare il lavoro in *team*, viene inoltre utilizzato *Atlassian Confluence*, uno strumento utile per raccogliere e organizzare documentazione e procedure aziendali. Nel contesto dell'applicazione prodotta, questa piattaforma è stata utilizzata per documentare i processi di creazione e modifica di un *tenant*, in modo da rendere chiari i vari passaggi durante lo sviluppo del prodotto.

1.4.2 Tecnologie e strumenti per la gestione di progetto

Per la gestione dei progetti, l'azienda si affida a *Git* e *GitHub*, strumenti fondamentali per il controllo di versione del codice e la collaborazione durante il processo di sviluppo. I punti di forza di *GitHub* risiedono nell'integrazione con numerosi strumenti DevOps$_G$, utili per configurare funzionalità di Continuous Integration (CI)$_G$ e Continuous Deployment (CD)$_G$, e nella capacità di ospitare sia progetti open source$_G$ che privati, come nel caso delle repository$_G$ di *Datasoil*.

Atlassian Jira viene invece utilizzato come Issue Tracking System (ITS) per il tracciamento dei *bug*, la pianificazione delle attività e il monitoraggio dello stato di avanzamento dei progetti.

1.4.3 Tecnologie e strumenti di sviluppo

Per le attività di codifica e testing è stato utilizzato Visual Studio Code (VS Code), un editor di codice *open source* con supporto per il debugging$_G$ e una vasta gamma di estensioni che si integrano con strumenti esterni e linguaggi di programmazione, migliorando il processo di sviluppo e aumentando la produttività.

Per facilitare lo sviluppo ed evitare conflitti tra le diverse configurazioni *hardware* dei dipendenti, l'azienda utilizza *Docker*, uno strumento in grado di garantire portabilità e riproducibilità in tutti gli ambienti di esecuzione del *software*.

Datasoil si affida alla piattaforma Amazon Web Services (AWS), con servizi come *Cognito*, Elastic Compute Cloud (EC2), Elastic Container Service (ECS) e Simple Storage Service (S3), per assicurare scalabilità e affidabilità nel rilascio e nell'hosting$_G$ delle applicazioni sviluppate. Questi strumenti verranno approfonditi in maniera più esaustiva in (§ 7, Integrazione e distribuzione continue), in cui verranno riportate tutte le interazioni tra le tecnologie che costituiscono il processo di rilascio.

Per implementare la componente backend$_G$ è stato utilizzato *GoLang*, un linguaggio di programmazione compilato noto per le sue elevate prestazioni. D'altra parte, l'impiego di *React* e *TypeScript* hanno permesso di realizzare la componente frontend$_G$ tramite la produzione di interfacce utente interattive e robuste. Infine, la persistenza dei dati è stata resa possibile grazie all'utilizzo di *MongoDB*, un Database Management System (DBMS)$_G$ non relazionale orientato ai documenti.

1.5 Struttura dello stage e riepilogo delle attività svolte

Lo stage ha avuto una durata totale di 300 ore, svolte in modalità ibrida, otto ore al giorno, dal 6 maggio al 26 giugno. In particolare, i primi tre giorni di ogni settimana sono stati svolti in presenza, mentre i restanti due da remoto.

1.5.1 Pianificazione

Lo stage è stato strutturato in periodi settimanali con obiettivi precisi, riportati qui in seguito.

Per contestualizzare i quantitativi orari dichiarati nel consuntivo finale, la lista delle attività svolte per ogni settimana viene riportata con maggiore precisione in (§ 8.1, Consuntivo finale).

Periodo	Attività
6 maggio - 10 maggio	Introduzione nel contesto aziendale, analisi tecnologica e dei requisiti.
13 maggio - 17 maggio	Progettazione e codifica della struttura delle componenti di alto livello.
20 maggio - 24 maggio	Implementazione del sistema di autenticazione e delle componenti *backend* e *frontend*.
27 maggio - 31 maggio	Configurazione della *pipeline* di rilascio e dei vari ambienti *Docker*.
3 giugno - 7 giugno	*Testing* e documentazione.
10 giugno - 14 giugno	Studio e progettazione delle *dashboard*.
17 giugno - 21 giugno	Implementazione delle *dashboard*.
24 giugno - 26 giugno	Revisione del lavoro svolto.

Tabella 1.1: Pianificazione delle attività.

1.6 Struttura della relazione

La presente relazione è strutturata in modo da seguire le attività in ordine di svolgimento, creando un filo conduttore che collega le prime attività di analisi e raccolta dei requisiti al rilascio del prodotto finale. Ogni capitolo rappresenta in questo senso un'attività o un processo fondamentale del ciclo di vita di un prodotto *software*.

Il testo è organizzato nei seguenti capitoli:

Secondo capitolo, "Analisi dei requisiti" : elenca i requisiti del prodotto organizzandoli per categorie;

Terzo capitolo, "Analisi tecnologica" : descrive ogni tecnologia analizzata, riportando pro e contro e motivando le scelte finali;

Quarto capitolo, "Progettazione" : illustra e motiva l'architettura scelta seguendo un approccio *top-down*;

Quinto capitolo, "Dettagli implementativi" : elenca i principali aspetti implementativi, riportando frammenti di codice e descrivendo l'attività di realizzazione delle componenti progettate;

Sesto capitolo, "Verifica e validazione" : riporta le modalità di svolgimento delle attività di verifica e validazione ed elenca i vari tipi di test effettuati, includendo frammenti di codice;

Settimo capitolo, "Integrazione e distribuzione continue" : descrive gli strumenti e le procedure di configurazione che abilitano l'integrazione e la distribuzione continue;

Ottavo capitolo, "Conclusioni" : riassume le attività svolte, giudicando in maniera critica il prodotto ottenuto e valutando in retrospettiva il lavoro svolto.

1.7 Convenzioni tipografiche

Per facilitare la lettura e mantenere coerente lo stile tipografico lungo tutta il corso del documento, vengono definite le seguenti norme, che verranno rispettate in qualunque occasione, a meno di quando indicato diversamente:

- Elenchi: ogni punto termina con ";", fatta eccezione per l'ultimo che termina con ".";
- Riferimenti interni: menzioni e riferimenti a sezioni interne vengono riportate seguendo la notazione (§ Numero sezione, Nome sezione). Questi riferimenti saranno opportunamente collegati tramite link alla sezione indicata;
- Glossario: gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati verranno definiti nel glossario, riportato nella sezione iniziale del presente documento;
- Prima occorrenza dei vocaboli del glossario: per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente notazione: parola$_G$;
- Formato dei termini in lingua straniera: i termini in lingua straniera sono riportati in carattere *corsivo*;
- Formato dei termini informatici: i vocaboli che fanno riferimento a variabili, funzioni e altri costrutti informatici sono riportati in carattere `monospaziato`.

Capitolo 2

Analisi dei requisiti

Lo scopo dell'attività di analisi dei requisiti è comprendere e definire in modo completo e accurato le esigenze e le aspettative del richiedente e degli utenti relativamente al prodotto *software*.

Questa attività è stata svolta durante la prima settimana assieme al reparto *backend*, contemporaneamente a sessioni di studio del prodotto *SYN* e delle sue funzionalità. Partendo dall'idea iniziale e dalle funzionalità di base individuate nella produzione del piano di lavoro, sono state raccolte tutte le richieste principali. Ognuna di queste è stata poi approfondita e scomposta, in modo da rendere i requisiti più granulari, facilitando il tracciamento di questi nelle attività di progettazione e codifica.

2.1 Analisi preliminare

2.1.1 Obiettivi del prodotto

Il prodotto deve offrire agli utenti la possibilità di interagire con le configurazioni dei diversi *tenant* legati ai vari clienti, permettendo la creazione, la modifica e la supervisione di essi. L'applicazione web deve inoltre fungere da strumento di analisi statistica e commerciale della piattaforma *SYN*, mostrando grafici e *dashboard* in grado di fornire informazioni utili per il tracciamento dell'utilizzo del prodotto da parte dei clienti nel corso del tempo.

2.1.2 Caratteristiche del prodotto

L'applicazione web dovrà essere accessibile solamente a un gruppo ristretto di utenti. L'interfaccia grafica dovrà essere familiare e intuitiva, offrendo una visione d'insieme delle informazioni principali di tutti i *tenant* e un accesso immediato ad una visione più specifica e precisa per ogni singolo *tenant*. Le funzionalità offerte dovranno essere robuste e progettate in modo da gestire una grande quantità di dati garantendo isolamento e correttezza.

2.1.3 Target di riferimento

Il prodotto è un *tool* interno rivolto esclusivamente ai dipendenti dell'azienda che si occupano della gestione dei clienti e delle loro configurazioni dal punto di vista informatico. Dato il preciso *target* di riferimento, la progettazione e lo sviluppo di questa applicazione presuppongono che l'utente conosca a priori l'architettura del prodotto *SYN* e i concetti informatici trattati.

2.2 Lista dei requisiti

I requisiti finali vengono riportati nelle sezioni sottostanti.

In queste sezioni verranno citati spesso i termini *my tenant* e *master tenant*, che costituiscono due modalità per riferirsi al reale concetto di *tenant* nel contesto dell'applicazione sviluppata. Come verrà spiegato approfonditamente in (§ 4.3.3, Sistemi di persistenza), le principali informazioni riguardanti le configurazioni dei *tenant* sono raccolte in un *database* unico, detto *master*, formando il concetto di *master tenant*, mentre le informazioni e i dati di *SYN* di ogni *tenant* vengono salvate in *database* diversi, unici per ogni *tenant*, rappresentando il concetto di *my tenant*.

2.2.1 Notazione

Ogni requisito preso in analisi sarà riportato utilizzando la seguente sigla:
R[Tipo][Importanza].[Codice], dove:

R : indica la parola "requisito";

Tipo : può assumere i valori:

- F (Funzionale);
- Q (Qualità);
- V (Vincolo).

Importanza : può assumere i valori:

- O (Obbligatorio);
- D (Desiderabile);
- OP (Opzionale).

Codice : identifica i requisiti per ogni tipologia. In caso di requisito padre, è composto da un numero progressivo univoco. In caso di sotto-requisito, segue il formato [CodicePadre].[CodiceFiglio]; questa struttura è ricorsiva, quindi non pone un limite alla profondità della gerarchia.

2.2.2 Requisiti funzionali

I requisiti funzionali descrivono le funzionalità del sistema, le azioni che il sistema può compiere e le informazioni che il sistema può fornire.

Codice	Descrizione
RF.O.1	L'utente deve poter accedere al servizio.
RF.O.1.1	L'utente deve poter inserire le proprie credenziali tramite l'autenticazione federata di *Google*.
RF.O.1.2	L'applicazione deve mostrare un errore all'utente nel caso in cui le credenziali inserite non siano valide.
RF.O.2	L'utente deve poter uscire dal proprio *account*.

Continua nella pagina seguente ...

Tabella 2.1 - Continuo della tabella

Codice	Descrizione
RF.O.3	L'utente deve poter creare un *tenant*.
RF.O.3.1	L'utente deve poter inserire le informazioni relative ad un *tenant* da memorizzare nel *database* comune.
RF.O.3.1.1	L'utente deve poter inserire le informazioni relative ad un *tenant* da memorizzare nella collezione *tenants* del *database* comune.
RF.O.3.2	L'utente deve poter inserire le informazioni relative ad un *tenant* da memorizzare nel suo *database* specifico.
RF.O.3.2.1	L'utente deve poter inserire le informazioni relative ad un *tenant* da memorizzare nella collezione *mytenant* del suo *database* specifico.
RF.O.3.2.2	L'utente deve poter inserire le informazioni relative ad un *tenant* da memorizzare nella collezione *sync_modules* del suo *database* specifico.
RF.O.3.3	L'applicazione deve mostrare un errore all'utente nel caso in cui le informazioni inserite non siano valide.
RF.O.4	L'applicazione deve poter inizializzare i ruoli di default durante la creazione di un *tenant*.
RF.O.5	L'utente deve poter modificare un *tenant*.
RF.O.5.1	L'utente deve poter modificare le informazioni relative ad un *tenant* da memorizzare nel *database* comune.
RF.O.5.1.1	L'utente deve poter modificare le informazioni relative ad un *tenant* da memorizzare nella collezione *tenants* del *database* comune.
RF.O.5.2	L'utente deve poter modificare le informazioni relative ad un *tenant* da memorizzare nel suo *database* specifico.
RF.O.5.2.1	L'utente deve poter modificare le informazioni relative ad un *tenant* da memorizzare nella collezione *mytenant* del suo *database* specifico.
RF.O.5.2.2	L'utente deve poter modificare le informazioni relative ad un *tenant* da memorizzare nella collezione *sync_modules* del suo *database* specifico.
RF.O.5.3	L'applicazione deve mostrare un errore all'utente nel caso in cui le informazioni modificate non siano valide.
RF.O.6	L'utente deve poter visualizzare la lista dei *tenant*.
RF.O.6.1	L'utente deve poter visualizzare le informazioni principali per ogni *tenant* in lista.
RF.O.7	L'utente deve poter visualizzare le informazioni relative ad un *tenant*.
RF.O.7.1	L'utente deve poter visualizzare le informazioni relative ad un *tenant* memorizzate nel *database* comune.
RF.O.7.1.1	L'utente deve poter visualizzare le informazioni relative ad un *tenant* memorizzate nella collezione *tenants*.

Continua nella pagina seguente ...

Tabella 2.1 - Continuo della tabella

Codice	Descrizione
RF.O.7.2	L'utente deve poter visualizzare le informazioni relative ad un *tenant* memorizzate nel *database* specifico.
RF.O.7.2.1	L'utente deve poter visualizzare le informazioni relative ad un *tenant* memorizzate nella collezione *mytenant*.
RF.O.7.2.2	L'utente deve poter visualizzare le informazioni relative ad un *tenant* memorizzate nella collezione *sync_modules*.
RF.O.8	L'utente deve poter visualizzare la lista degli indici.
RF.O.8.1	L'utente deve poter visualizzare le informazioni principali per ogni indice in lista.
RF.O.9	L'utente deve poter visualizzare le informazioni relative ad un indice.
RF.O.9.1	L'utente deve poter visualizzare le informazioni relative alle chiavi dell'indice.
RF.O.9.2	L'utente deve poter visualizzare le informazioni relative alle opzioni dell'indice.
RF.O.10	L'utente deve poter visualizzare la lista degli indici attivati per uno specifico *tenant*.
RF.O.11	L'utente deve poter attivare un insieme di indici per uno specifico *tenant*.
RF.O.12	L'utente deve poter disattivare un insieme di indici per uno specifico *tenant*.
RF.O.13	L'utente deve poter applicare un indice a tutti i *tenant* presenti.
RF.O.14	L'utente deve poter visualizzare il numero di utenti di un *tenant*.
RF.D.15	L'utente deve poter visualizzare un grafico che mostra il numero di utenti presenti in certo periodo di tempo.
RF.O.16	L'utente deve poter visualizzare il numero di *asset* di un *tenant*.
RF.D.17	L'utente deve poter visualizzare un grafico che mostra il numero di *asset* presenti in certo periodo di tempo.
RF.O.18	L'utente deve poter visualizzare il numero di *ticket* di un *tenant*.
RF.D.19	L'utente deve poter visualizzare un grafico che mostra il numero di *ticket* presenti in certo periodo di tempo.
RF.O.20	L'utente deve poter visualizzare il numero di *inspection* di un *tenant*.
RF.D.21	L'utente deve poter visualizzare un grafico che mostra il numero di *inspection* presenti in certo periodo di tempo.
RF.O.22	L'utente deve poter visualizzare il numero di *geocontainer* di un *tenant*.
RF.O.22.1	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Compound di un *tenant*.
RF.O.22.2	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Building di un *tenant*.

Continua nella pagina seguente ...

Tabella 2.1 - Continuo della tabella

Codice	Descrizione
RF.O.22.3	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Floor di un *tenant*.
RF.O.22.4	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Room di un *tenant*.
RF.OP.22.5	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Region di un *tenant*.
RF.OP.22.6	L'utente deve poter visualizzare il numero di *geocontainer* di tipo Subregion di un *tenant*.
RF.OP.22.7	L'utente deve poter visualizzare il numero di *geocontainer* di tipo External di un *tenant*.
RF.OP.22.8	L'utente deve poter visualizzare il numero di *geocontainer* di tipo SubExternal di un *tenant*.

Tabella 2.1: Lista dei requisiti funzionali.

2.2.3 Requisiti di qualità

I requisiti di qualità descrivono come un sistema deve essere, o come il sistema deve esibirsi, per soddisfare le esigenze dell'utente.

Codice	Descrizione
RQ.O.1	L'applicazione deve gestire gli errori in modo robusto.
RQ.O.2	L'applicazione deve garantire la segretezza dei dati, oscurando tutte le funzionalità agli utenti non autorizzati.

Tabella 2.2: Lista dei requisiti di qualità.

2.2.4 Requisiti di vincolo

I requisiti di vincolo descrivono i limiti e le restrizioni normative e legislative che un sistema deve rispettare per soddisfare le esigenze dell'utente.

Codice	Descrizione
RV.O.1	L'applicazione deve soddisfare gli obiettivi minimi definiti nel piano di progetto.
RV.O.2	La componente *backend* deve essere sviluppata utilizzando il linguaggio *GoLang*.
RV.O.3	La componente *frontend* deve essere sviluppata utilizzando la libreria *React*.
RV.O.4	La componente *frontend* deve essere sviluppata utilizzando il linguaggio *Typescript*.
RV.O.5	La persistenza dei dati deve essere realizzata utilizzando *database MongoDB*.

Continua nella pagina seguente ...

Tabella 2.3 - Continuo della tabella

Codice	Descrizione
RV.O.6	Il processo di rilascio deve essere eseguito tramite l'utilizzo di *Docker*.

Tabella 2.3: Lista dei requisiti di vincolo.

Capitolo 3

Analisi tecnologica

Lo scopo dell'attività di analisi tecnologica è analizzare le caratteristiche, i pregi e i difetti delle opzioni tecnologiche disponibili, effettuando scelte tra di loro compatibili che nell'insieme formano una soluzione che permette di implementare il prodotto richiesto in efficienza ed efficacia.

3.1 Esplorazione tecnologica

Assieme all'azienda è stato deciso di mantenere invariate la maggior parte delle tecnologie utilizzate in *SYN* anche per il progetto di stage. Questa scelta deriva dalla volontà di voler sviluppare un'applicazione web graficamente simile a *SYN* e facilmente manutenibile, in modo che cambiamenti apportati al sistema principale potessero essere agevolmente riflessi nel sistema di amministrazione.

Molte delle tecnologie scelte da *Datasoil* sono infatti usate ampiamente nei vari prodotti dell'azienda e rappresentano in generale scelte valide e consolidate nello sviluppo di applicazioni web moderne.

Per altri tipi di tecnologie, invece, l'azienda ha deciso di sfruttare questo progetto anche per esplorare le varie offerte disponibili sul mercato, individuando opzioni che potrebbero in futuro sostituire le attuali soluzioni adottate.

3.1.1 Module Bundler

Le interfacce grafiche e la logica di interazione con l'utente, che nel loro complesso costituiscono la componente *frontend*, sono state implementate tramite l'utilizzo di *React*, una libreria *JavaScript*, e *Typescript*, un superset di *JavaScript* sviluppato da *Microsoft* che aggiunge tipizzazione statica. L'utilizzo combinato di React e TypeScript permette di creare applicazioni web interattive, mantenendo al contempo un codice più robusto, manutenibile e scalabile, permettendo a *SYN* di gestire ogni giorno una grande mole di dati.

Nel contesto dello sviluppo *frontend*, un *module bundler* è uno strumento specializzato che automatizza il processo di organizzazione, ottimizzazione e combinazione di più file *JavaScript*, fogli di stile *CSS* e altre risorse in uno o più file di *output*. Questo processo è noto come "*bundling*", o raggruppamento, e viene eseguito per ottimizzare il caricamento e l'esecuzione di un'applicazione web all'interno di un ambiente *browser*. Poiché le applicazioni web continuano a crescere in complessità, aumenta anche il numero di risorse e dipendenze al loro interno. I *bundler* di moduli *frontend* eseguono diverse attività fondamentali, tra cui la risoluzione delle relazioni tra moduli e dipendenze e la riduzione del numero di richieste

e delle dimensioni complessive delle risorse fornite al *browser*, migliorando di conseguenza le prestazioni dell'applicazione web.

Nel prodotto *SYN* viene utilizzato *Rollup*, uno dei *bundler* più popolari nel mondo *Javascript*. *Rollup* è in grado di migliorare la gestione delle risorse e diminuire significativamente le dimensioni dei *bundle* prodotti, analizzando i grafi delle dipendenze e utilizzando avanzate tecniche di *code-splitting* e *tree shacking*, cioè tecniche di ottimizzazione che permettono di analizzare ed eliminare codice inutilizzato. Questo *bundler* è particolarmente adatto in situazioni in cui l'applicazione web dipende da un grande numero di pacchetti e moduli esterni.

Tuttavia, nello sviluppo di applicazioni più semplici e con un numero limitato di dipendenze esterne, *Rollup* risulta più complesso da configurare e gestire rispetto ad altre soluzioni più moderne.

La scelta finale è quindi ricaduta sull'alternativa *Vite*, un *bundler* progettato per fornire un'esperienza di sviluppo semplice e rapida, fornendo soluzioni pre-configurate per i principali framework$_G$ web. A differenza di *Rollup*, *Vite* utilizza due modalità di *bundling* diverse per gli ambienti di sviluppo e produzione. Nell'ambiente di sviluppo, *Vite* è ottimizzato per aumentare la velocità di compilazione, mentre nell'ambiente di produzione assicura maggiori prestazioni a runtime$_G$, a discapito di un tempo di compilazione e *bundling* più lungo. *Vite* include inoltre un *server* di sviluppo in grado di mantenere invariato lo stato dell'applicazione anche durante l'aggiornamento del codice e delle dipendenze, velocizzando di molto i tempi di sviluppo.

3.1.2 Libreria per l'implementazione delle dashboard

Per creare *dashboard* interattive e visivamente accattivanti in *React*, è fondamentale scegliere una libreria grafica in linea con le proprie esigenze, cercando di bilanciare funzionalità, facilità d'uso e prestazioni.

All'interno del prodotto *SYN*, le *dashboard* vengono implementate attraverso un servizio apposito interno, altamente configurabile e in grado di gestire in contemporanea grafici complessi. Nonostante questo, per poter utilizzare questo servizio sarebbe stato necessario effettuare una configurazione iniziale non immediata. Sono state quindi considerate alternative esterne, meno personalizzabili ma più facilmente integrabili all'interno del progetto.

Uno degli strumenti più utilizzati nell'ambiente *Javascript* per la creazione di grafici è *D3.js*, una libreria in grado di generare visualizzazioni dinamiche e interattive concedendo un buon livello di controllo sulla resa visiva finale. Nonostante la sua flessibilità, *D3.js* ha una curva di apprendimento piuttosto ripida e può risultare pesante in termini di dimensioni e prestazioni.

Tra le soluzioni più leggere e facili da utilizzare, la più popolare è *Chart.js*, una libreria *open source* seconda in termini di utilizzo, inferiore solo a *D3.js*. L'integrazione con la libreria grafica *PrimeReact*, utilizzata da *Datasoil*, e la sua facilità d'uso hanno reso *Chart.js* la soluzione ideale per il progetto, permettendo di implementare rapidamente grafici efficaci senza compromettere le prestazioni dell'applicazione.

3.1.3 Backend web framework

GoLang, o semplicemente *Go*, è un linguaggio di programmazione *open source* sviluppato da *Google*. È noto per la sua efficienza e velocità ed è particolarmente adatto per lo sviluppo di applicazioni *server-side*. È stato progettato per garantire una compilazione efficiente, alte prestazioni e una bassa soglia di complessità sintattica. *Datasoil* utilizza *Go* come linguaggio

di *backend* in tutti i suoi prodotti, poiché riesce a soddisfare le esigenze di robustezza e scalabilità, mantenendo buone prestazioni anche in situazioni di elevato traffico.

Una buona parte della prima settimana è stata dedicata allo studio e all'analisi dei vari *web framework* basati sul linguaggio *Go* disponibili sul mercato.

Agli inizi dello sviluppo *software* in *Go*, la base per poter creare un'applicazione in grado di esporre Application Program Interface (API)$_G$ era costituita dal package *net/http*, appartenente alla libreria standard. Con l'avanzamento degli anni, come per ogni altro linguaggio di programmazione, la *community* ha iniziato a creare astrazioni, sotto forma di librerie o *web framework*, che facilitano lo sviluppo di queste applicazioni, migliorando le *performance* e offrendo un maggior numero di funzionalità. Nei prodotti di *Datasoil* viene utilizzato *Negroni*, una libreria non intrusiva e molto leggera, progettata per interfacciarsi direttamente con gli *handlers* del *package net/http*. *Negroni* è stata una delle prime librerie sviluppate nel linguaggio *Go*, e per questo motivo soffre di alcune mancanze che rendono lo sviluppo di *API* molto più lento e verboso rispetto a soluzioni più recenti. Uno degli obiettivi di questo stage era quindi anche individuare un'alternativa a *Negroni*, che permettesse a *Datasoil* di migliorare non solo le prestazioni dei loro prodotti, ma anche la qualità del loro codice sorgente. Prima di iniziare l'esplorazione, sono stati individuati assieme ai dipendenti del reparto *backend* i punti carenti di *Negroni* e le funzionalità base che la scelta finale avrebbe dovuto garantire, oltre a richieste *extra* "*nice to have*", che avrebbero determinato il vincitore nel caso in cui le richieste di base non fossero state sufficienti per individuare un'unica scelta finale.

Caratteristiche ricercate

Le principali caratteristiche ricercate nei vari *web framework* disponibili erano le seguenti:

- Binding: presenza di uno strumento in grado di effettuare un collegamento fra un'entità *software* e il suo corrispettivo valore;
- Validazione: presenza di uno strumento per la validazione dei dati in entrata tramite regole di controllo e correttezza;
- *Marshalling* e *unmarshalling*: presenza di uno strumento capace di effettuare conversioni tra diverse rappresentazioni di un'entità *software*;
- Bassa astrazione: possibilità di interagire con le principali componenti di basso livello del *framework*, permettendo una personalizzazione significativa del comportamento e delle caratteristiche delle varie componenti. Data la complessità di *SYN*, il *framework* deve permettere una configurazione ampia e di basso livello. Questa caratteristica è spesso assente nei *web framework* più semplici e astratti;
- Minor verbosità rispetto a *Negroni*: interfacciandosi direttamente con i componenti base della libreria standard *net/http*, *Negroni* risulta piuttosto verboso e prolisso, anche in situazioni comuni e non complesse;
- Possibilità di servire file statici: possibilità di esporre file e cartelle, rendendole disponibili attraverso apposite *route API*;
- *Middleware*: capacità di implementare *middleware*, componenti *software* che eseguono una funzione o offrono un servizio nel momento in cui il sistema riceve una chiamata dall'ambiente esterno.

Framework esplorati

Complessivamente, sono stati analizzati i seguenti *web framework*:

- *Gin*: è il *web framework* più popolare nel linguaggio *Go*. È noto per la sua velocità e semplicità. Una delle sue caratteristiche principali è l'uso di un *middleware* per la gestione delle richieste, che consente di creare facilmente catene di operazioni da eseguire su ogni richiesta HyperText Transfer Protocol (HTTP)$_G$. *Gin* offre funzionalità di *binding* e validazione integrate, permettendo di associare direttamente i dati delle richieste a strutture *Go* e di verificarne la correttezza, offrendo ampia scelta e regole di validità anche piuttosto articolate. Inoltre, supporta il *marshalling* e *unmarshalling* tramite JavaScript Object Notation (JSON), eXtensible Markup Language (XML) e altre rappresentazioni, semplificando la gestione dei dati. La sintassi di *Gin* è meno verbosa di *Negroni*, mantenendo comunque un'astrazione di basso livello. Permette la gestione e la pubblicazione di file statici attraverso la semplice esposizione di apposite *route*. La sua grande popolarità si riflette nella qualità della sua documentazione e nella grande quantità di risorse informative, oltre al buon supporto della *community*;

- *Echo*: *Echo* è un *web framework* progettato per essere veloce e facile da usare. Fornisce strumenti per il *binding* e la validazione dei dati delle richieste, semplificando il processo di associazione e verifica dei dati. Supporta *marshalling* e *unmarshalling* per diverse rappresentazioni di dati, facilitando la conversione tra formati come *JSON* e *XML*. È meno verboso di *Negroni* e offre una gestione semplificata delle rotte e dei *middleware*. Include anche il supporto per la gestione di file statici. La sua semplicità d'uso limita però la personalizzazione della configurazione del *framework*, risultando in un'astrazione eccessiva rispetto a quella ricercata dall'azienda;

- *Fiber*: *Fiber* è un *framework* noto per le sue ottime prestazioni e per la sua facilità d'uso. Queste caratteristiche rispecchiano la sua continua crescita negli ultimi anni, che l'ha portato ad essere il secondo *framework* più utilizzato nell'ecosistema *Go*, inferiore solo a *Gin*. La sua *community* è molto attiva e disponibile. Offre funzionalità di *binding* e validazione dei dati, seppur con possibilità più limitate rispetto ai *framework* sopracitati. Il *framework* supporta *marshalling* e *unmarshalling* per diverse rappresentazioni di dati e include il supporto per servire file statici. La sua sintassi è molto meno verbosa rispetto a *Negroni*, migliorando la leggibilità e la manutenibilità del codice. Come appurato per il *framework* Echo, la sua astrazione è risultata eccessiva e probabilmente più adatta a progetti di dimensioni e complessità minori;

- *Mux*: *Gorilla Mux* è un *router* potente e flessibile. *Mux* fornisce supporto limitato per il *binding* dei dati delle richieste e richieste spesso librerie aggiuntive per la validazione e il *marshalling* e *unmarshalling* dei dati. Questo *framework* offre un buon grado di controllo sulle componenti di basso livello, permettendo una personalizzazione significativa del comportamento delle rotte e dei *middleware*. Rispetto alle alternative citate sopra, *Mux* risulta più verboso. Inoltre, la possibilità di servire file statici non è direttamente inclusa nel *framework* ma può essere disposta attraverso l'uso di altre librerie esterne. In generale, *Mux* assomiglia molto a *Negroni*, dalla quale l'azienda si vuole allontanare;

Durante l'analisi delle varie opzioni, anche attraverso lo sviluppo di piccoli esempi dimostrativi, è stato possibile raggruppare i *framework* individuati in base alle caratteristiche ricercate. In particolare, è stata subito individuata una netta separazione in base al livello

di astrazione. Questa caratteristica, più delle altre, è stata cruciale per effettuare la scelta finale, essendo prettamente legata alla struttura e all'idioma del *framework*.

Dopo aver illustrato e discusso con il reparto *backend* i pro e i contro di ogni *framework* analizzato, la scelta finale è ricaduta sul *web framework Gin*, l'unico in grado di soddisfare tutti i requisiti richiesti dall'azienda.

Capitolo 4

Progettazione

L'attività di progettazione ha lo scopo di determinare un'architettura che soddisfi i requisiti determinati dall'attività di analisi, prima di passare alla codifica. La progettazione serve a dominare la complessità del prodotto, decomponendolo in parti componibili e organizzate.

4.1 Architettura infrastrutturale

Per poter capire meglio le scelte progettuali fatte in questo progetto, è necessario introdurre l'architettura esistente del prodotto *SYN*, su cui l'applicazione oggetto di stage si deve appoggiare.

SYN è basato su un'architettura *SaaS multi-tenant*.

SaaS è un modello di servizio di cloud computing$_G$ per la quale un produttore mette a disposizione un *software* offrendolo ai suoi clienti tramite modalità telematica, sfruttando ad esempio la rete *Internet*.

In questo caso, il *software* non è installato localmente, ma è distribuito tramite il *cloud*. Il *cloud computing* consente ai produttori di *software* di adoperare risorse di *computing* su *Internet* come un'*utility*, rimuovendo l'onere di dover creare e gestire l'infrastruttura fisica necessaria per poter eseguire e distribuire il proprio prodotto.

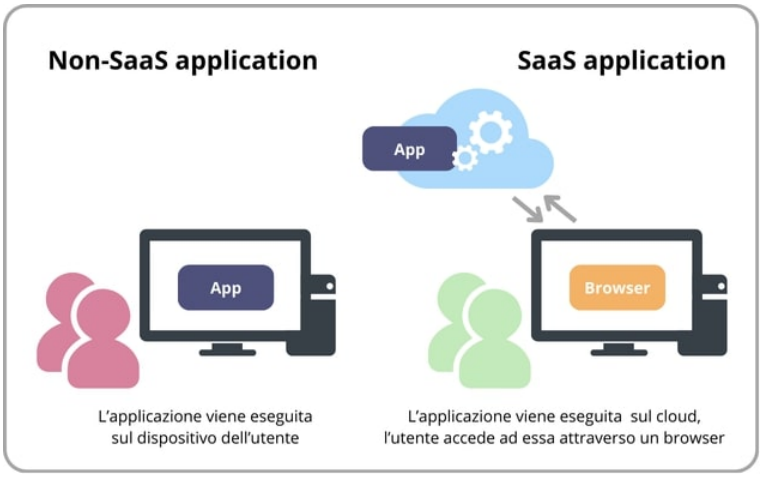

Figura 4.1: Confronto tra *Non-SaaS* e *SaaS*

I principali vantaggi di *SaaS* sono i seguenti:

- Costi di configurazione e infrastruttura ridotti: il modello *SaaS* è generalmente offerto con un prezzo di abbonamento basato sul numero di utenti o sui volumi di dati gestiti. Questo approccio riduce la necessità di investimenti iniziali in *hardware* e manutenzione, comportando una diminuzione dei costi operativi;

- Accesso semplificato: dal punto di vista del cliente, il *software* non è vincolato a una macchina fisica specifica e può essere accessibile da qualsiasi dispositivo in grado di raggiungere la rete sulla quale il servizio è distribuito. Questo consente una maggiore flessibilità di accesso per gli utenti;

- Scalabilità: le soluzioni *SaaS* offrono una scalabilità facilitata, permettendo di adeguare rapidamente le risorse in base alle esigenze degli utenti. Questo processo risulta meno dispendioso e tecnicamente meno complesso rispetto alle infrastrutture tradizionali;

- Aggiornamenti automatici e frequenti: il *provider* dell'applicazione è responsabile degli aggiornamenti e delle correzioni. Le nuove funzioni e correzioni vengono applicate in modo centralizzato, senza richiedere interventi da parte del cliente, migliorando l'efficienza operativa rispetto ai modelli *Non-SaaS*.

Uno dei concetti alla base di *SaaS* è che il *software* distribuito è formato da un singolo insieme di codice e definizione di dati comuni, che vengono utilizzati da più clienti in un modello uno-a-molti.

Per riuscire a gestire e isolare i dati dei clienti, le applicazioni *SaaS* sfruttano l'architettura *multi-tenant*. La *multitenancy* è un'architettura in cui una singola istanza *software* è in grado di servire più gruppi di utenti distinti, divisi in *tenant*. Un'applicazione *multi-tenant* offre personalizzazione a livello di singolo *tenant*, mentre l'intera architettura e le funzionalità principali rimangono comuni.

I concetti base di questa architettura si possono ritrovare nel timesharing$_G$ degli anni '60, in cui le aziende affittavano spazio e potenza di calcolo sui *mainframe*, e nel concetto di Application Service Provider (ASP), per la quale un fornitore di servizi ospita applicazioni per conto dei clienti. Attualmente, questa architettura è principalmente utilizzata nelle applicazioni web, distribuite come singola istanza, usufruibile da più utenti.

La gestione dei dati in un modello *multi-tenant* può essere affrontata nei seguenti modi:

- Singolo *database* e singolo schema: un unico *database* contiene i dati di tutti i *tenant*. In questo caso, la separazione dei dati è garantita tramite l'uso di identificativi di *tenant*, che aiutano a distinguere le varie *entry*. Questa soluzione è facile da manutenere ma è poco flessibile;

- Singolo *database* e schemi multipli: tramite l'uso di più schemi, è possibile definire tabelle, procedure e permessi per diversi clienti. Questa soluzione è più flessibile della precedente ma divide i dati solo a livello logico, rendendo problematici i processi di *backup* e recupero;

- *Database* separati: ogni *tenant* è associato ad un database diverso, in cui vengono memorizzati esclusivamente i suoi dati. Questo approccio è il più flessibile e sicuro, offre una buona scalabilità e facilita i processi di *backup* e recupero. È la soluzione più costosa in termini di risorse.

La differenza principale tra segregazione logica e fisica è il modo in cui l'isolamento è applicato. Quando più *tenant* condividono un'infrastruttura, l'isolamento dei dati deve essere

garantito a livello del codice dell'applicazione. In questo caso, il livello logico simula la completa separazione di dati e risorse. Al contrario, quando i *tenant* sono separati a livello fisico, cioè ognuno ha una sua infrastruttura dedicata, il codice dell'applicazione non deve necessariamente essere progettato per operare in un ambiente *multi-tenant*, poiché l'isolamento è garantito dall'infrastruttura stessa.

La *multitenancy* riduce i costi delle risorse e porta beneficio nella gestione dei rilasci e nei costi di manutenzione. D'altro canto, l'utilizzo di questa architettura in un contesto non adatto porta a complessità di sviluppo, flessibilità limitata, problemi di sicurezza e basse *performance*.

Questa architettura è particolarmente adottata da aziende *startup*, dati i bassi costi di entrata e l'immediatezza con cui si possono affrontare operazioni di manutenzione e correzione di *bug*.

Dovendo gestire clienti di diversa natura e con diverse necessità, *Datasoil* utilizza *database* separati, garantendo un buon livello di flessibilità e sicurezza, a discapito di un consumo di risorse maggiori. In generale, questa soluzione è la più adottata nel mondo *enterprise*.

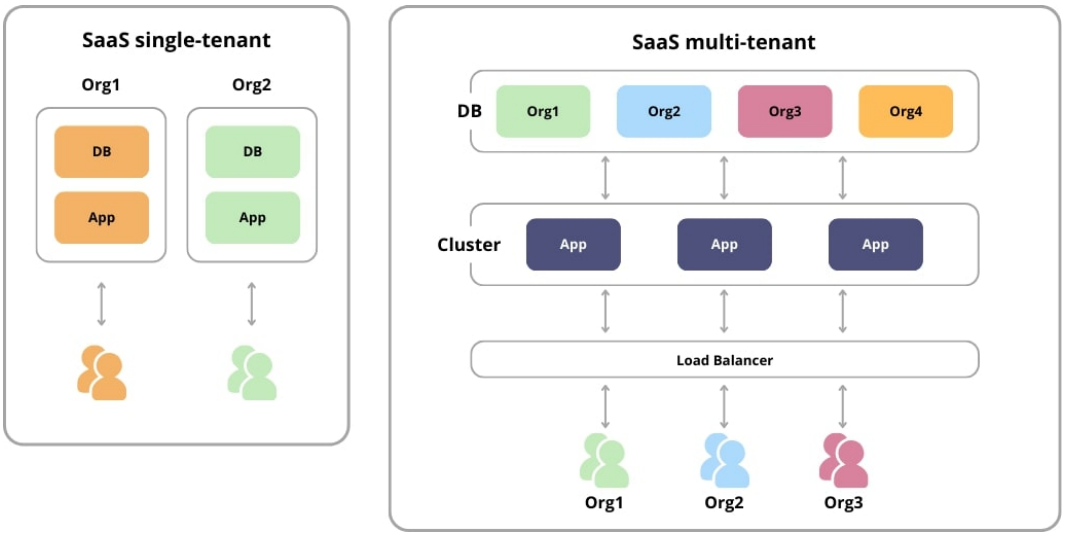

Figura 4.2: Schema dell'architettura *multi-tenant*

4.2 Architettura di sistema

Durante l'attività di progettazione, è stato deciso di mantenere l'architettura di sistema del progetto simile a quella utilizzata in *SYN*, così da facilitarne lo sviluppo e la manutenzione.

4.2.1 Modello architetturale

SYN è basato su un'architettura di sistema "*three-tier*", derivata dal gruppo delle architetture "*multi-tier*" o "*N-tier*". È un'architettura *software* di tipo *client-server*, in cui le varie funzionalità del *software* sono logicamente separate, ovvero suddivise su più strati o livelli *software* differenti in comunicazione tra loro.

Nelle architetture *N-tier*, ciascuno strato è in comunicazione diretta con quelli adiacenti, cioè richiede ed offre servizi agli strati vicini. In generale, i vari strati possono risiedere sulla stessa macchina oppure su macchine differenti, mappando l'applicazione a livello infrastrutturale.

L'architettura delle applicazioni *N-tier* fornisce un modello per creare un'applicazione flessibile e riutilizzabile, consentendo di apportare modifiche ad uno specifico livello senza coinvolgere il resto dell'applicazione.

Three-tier è un'architettura *client-server* in cui l'interfaccia utente, i processi logici ("business logic" o "regole aziendali") e l'archiviazione dei dati sono sviluppate e mantenute come moduli indipendenti.

È costituita dai seguenti livelli:

- Livello di presentazione: si occupa di organizzare e mostrare le informazioni richieste e offre all'utente la possibilità di interagire con l'applicazione;
- Livello applicazione: ha il compito di soddisfare le richieste provenienti dal livello di presentazione, verificando che le richieste siano sintatticamente e semanticamente corrette, eseguendo elaborazioni e interagendo con il livello dati sottostante;
- Livello dati: è costituito da sistemi di memorizzazione, come *database* o *file system*. Ha il compito di conservare e organizzare le informazioni in modo da renderle facilmente accessibili al livello applicazione.

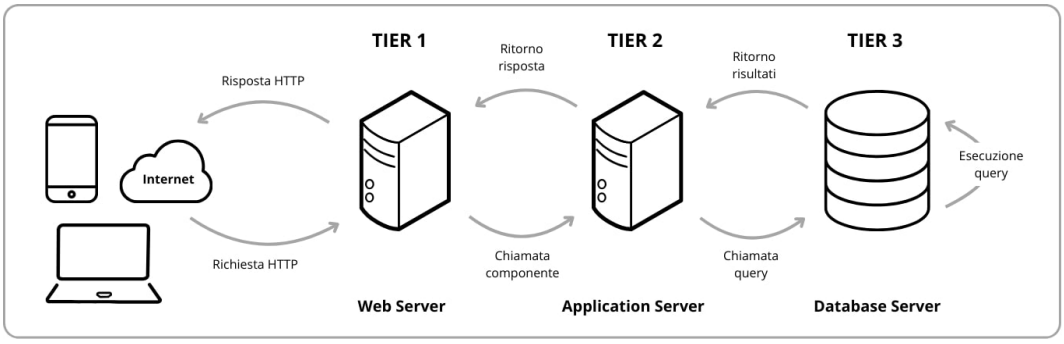

Figura 4.3: Architettura di sistema

4.2.2 Descrizione delle componenti

L'architettura *three-tier* è particolarmente utilizzata nello sviluppo di applicazioni web, spesso composte dalle seguenti componenti:

- *Frontend*: rappresenta il livello di presentazione. Nel contesto web, l'utente accede ad esso tramite un *browser*;
- *Backend*: rappresenta il livello applicazione. Offre i suoi servizi all'ambiente esterno tramite l'esposizione di *API*, un insieme di definizioni e contratti che svolge la funzione di intermediario tra due livelli, stabilendo il contenuto richiesto e le risposte fornite;
- Sistema di persistenza: costituisce il livello dati. Persiste informazioni degli utenti, delle loro richieste e i risultati delle elaborazioni effettuate dal *backend*.

4.2.3 Assemblaggio delle componenti

L'assemblaggio delle componenti è stato semplificato dall'utilizzo di *Docker*, che ha permesso di definire dei *container* isolati, uniti poi tramite lo strumento *Docker Compose*. Questo strumento consente di eseguire un'intera applicazione assemblando più *container* ed esponendo i relativi servizi in diverse porte, tramite le quali i *container* possono scambiare informazioni.

I vari ambienti di esecuzione configurati tramite *Docker Compose* verranno discussi in (§ 7, Integrazione e distribuzione continue).

4.3 Architettura delle componenti

Questa sezione elenca le varie sotto-componenti di *frontend*, *backend* e sistema di persistenza, illustrando le loro funzionalità.

4.3.1 Frontend

Sidebar

Questo componente si occupa di mostrare una barra di navigazione laterale che permette all'utente di accedere alle varie sezioni dell'applicazione web.

È formata da:

- `ProfileIcon`: mostra un *avatar* dell'utente;
- `SynMenuIcons`: rappresenta un insieme di pagine tramite un'icona esplicativa;
- `Menu`: raggruppa e organizza le varie pagine dell'applicazione in `SynMenuIcons`;
- `SynSidebar`: gestisce i componenti della *sidebar* e lo stato di *login* dell'utente.

Login

Questo componente offre all'utente la possibilità di autenticarsi tramite una pagina apposita che porta al servizio di autenticazione federata di *Google* tramite *AWS Cognito*.

Logout

Questo componente permette all'utente di scollegarsi dal proprio account, uscendo dall'applicazione web.

Home

Questo componente rappresenta la pagina iniziale del sistema e si occupa di mostrare le informazioni principali, permettendo di accedere alla pagina singola di ogni *tenant*.

È formata da:

- `SynHeader`: mostra il titolo della pagina;
- `TenantsTable`: componente che mostra una tabella contenente le informazioni principali dei *tenant* e un collegamento alla pagina di ogni singolo *tenant*.

TenantsView

Questo componente mostra le principali informazioni di ogni *tenant*, offrendo funzionalità di ricerca, filtraggio e ordinamento.

È formata da:

- `SynHeader`: mostra il titolo della pagina e un tasto di azione che porta alla creazione di un nuovo *master tenant*;
- `TenantsTable`: componente che mostra una tabella contenente le informazioni principali dei *tenant* e un collegamento alla pagina di ogni singolo *tenant*.

TenantView

Questo componente organizza e mostra tutte le informazioni relative a uno specifico *tenant* quando viene passato un id, altrimenti permette di creare un nuovo *tenant*.

È formata da:

- `SynHeader`: mostra il titolo della pagina e un tasto di azione che porta alla modifica del *tenant* in questione;
- `MasterTenantForm`: form che offre la possibilità di compilare le informazioni relative ad un *master tenant*;
- `MyTenantForm`: form che permette di compilare le informazioni relative ad un *my tenant*;
- `MainCountersSection`: sezione formata da tre componenti `StatsCounter` che mostrano il numero di utenti, *assets* e *tickets*;
- `MasterTenantInfoSection`: sezione che organizza le informazioni di un *master tenant*;
- `MyTenantInfoSection`: sezione che organizza le informazioni di un *my tenant*;
- `UsersStatsSection`: sezione che mostra, in base ai parametri impostati, un grafico rappresentante la progressione temporale degli utenti assieme alle statistiche principali, tra cui massimo, minimo, media e andamento;
- `EntitiesStatsSection`: sezione che mostra, in base ai parametri impostati, un grafico rappresentante le progressioni temporali sovrapposte di *assets*, *tickets* e *inspections*, assieme alle loro statistiche principali, tra cui massimo, minimo, media e andamento;
- `IndexesTable`: componente formato da una tabella che elenca gli indici attivi nel *tenant* in questione;
- `SyncModulesSection`: componente che mostra le informazioni relative ai moduli *sync*, tra cui il modulo *efm*;
- `GeoContainersStatsSection`: sezione che mostra un contatore per ciascun tipo di *geocontainer* selezionato.

EditTenantView

Questo componente permette di modificare ogni informazione relativa a uno specifico *tenant*.

È formata da:

- `SynHeader`: mostra il titolo della pagina;
- `MasterTenantForm`: consente di modificare il *master tenant* a partire dai suoi valori attuali;
- `MyTenantForm`: permette di modificare il *my tenant* a partire dai suoi valori attuali;
- `TenantIndexesForm`: mostra una tabella che riporta la lista di tutti gli indici, ciascuno affiancato da un selettore che consente di attivare o disattivare l'indice associato;
- `SyncModuleForm`: permette di modificare il modulo *efm* a partire dai suoi valori attuali;

IndexesView

Questo componente mostra le principali informazioni di ogni indice, offrendo funzionalità di ricerca, filtraggio e ordinamento.

È formata da:

- `SynHeader`: mostra il titolo della pagina;
- `IndexesTable`: componente che mostra una tabella contenente le informazioni principali degli indici e un collegamento alla pagina di ogni singolo indice.

IndexView

Questo componente organizza e mostra tutte le informazioni relative a uno specifico indice.

È formata da:

- `SynHeader`: mostra il titolo della pagina e un tasto di azione che consente di attivare l'indice in questione in tutti i *tenant*;
- `IndexOptionsSection`: sezione che mostra le informazioni relative alle opzioni dell'indice;
- `IndexKeysSection`: sezione che organizza in una tabella le informazioni relative alle chiavi dell'indice;

4.3.2 Backend

Le comunicazioni *API* in rete avvengono attraverso l'invio e la ricezione di richieste e risposte tra *client* e *server*. Tuttavia, l'ambiente della rete *Internet* è intrinsecamente inaffidabile e instabile a causa di vari fattori come la latenza, la perdita di pacchetti e problemi di connessione e congestione del traffico. Questi problemi possono portare a ritardi, errori o fallimenti nella trasmissione dei dati, rendendo fondamentale l'adozione di principi e pratiche che possano mitigare tali rischi.

Un modo per affrontare queste sfide è l'adozione dei principi Representational State Transfer (REST)$_G$, un insieme di linee guida che offrono un approccio strutturato e coerente per la comunicazione tra sistemi distribuiti.

Le *API* progettate seguono rigorosamente i seguenti principi *REST*:

- Stateless: ogni richiesta *API* è indipendente e contiene tutte le informazioni necessarie per essere processata. Questo significa che il server non mantiene alcuno stato tra le richieste, semplificando la gestione delle richieste;
- Interfaccia uniforme: tutte le *API* utilizzano un'interfaccia uniforme, definendo le modalità di interazione con le risorse, semplificando l'utilizzo delle *API* da parte di servizi esterni;
- Comunicazione in *JSON*: le *API* utilizzano il formato *JSON* per la trasmissione dei dati. *JSON* è leggibile sia dagli esseri umani che dalle macchine, il che facilita lo sviluppo, il *debugging* e l'integrazione con altri sistemi.

Ogni *route API* esposta all'esterno è inoltre protetta da un *middleware* di autorizzazione. Questo *middleware* verifica che in ogni richiesta in entrata al servizio sia presente un bearer token$_G$ valido, cioè una stringa criptata che viene fornita all'utente nel momento in cui viene effettuato correttamente il *login*.

Per evitare di ripeterla per ogni *route API* riportata in seguito, dalle tabelle degli esiti possibili viene omessa la risposta "*401, Unauthorized*", generata nel caso in cui la richiesta non contenga un *token* valido.

Inoltre, per evitare di essere troppo prolisso, è stata omessa la descrizione completa di alcune *route*, menzionandole solamente dopo la descrizione di *ruote* a loro simili.

Alive

Questa componente funge da indicatore di presenza del servizio *API*. Può essere usato da servizi esterni per verificare la correttezza della connessione al servizio.

È costituita da:

- *Route API*: `/master/alive`;
- Metodo: GET;
- *Body* della richiesta: vuoto.

Status	Descrizione	Risposta
200	Servizio *API* attivo.	"*I'm alive!*"

Tabella 4.1: Esiti possibili di Alive.

GetMasterTenants

Questa componente ha il compito di restituire la lista dei *master tenants*.

È costituita da:

- *Route API*: `/master/api/tenants/master`;
- Metodo: GET;
- *Body* della richiesta: vuoto.

Status	Descrizione	Risposta
200	Lista dei *master tenant* elaborata con successo.	Rappresentazione *JSON* della lista di oggetti `MasterTenant`.
500	Errore nell'elaborazione della lista a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.2: Esiti possibili di GetMasterTenants.

GetMyTenants

In modo simile a `GetMasterTenants`, è stata definita la *route* `GetMyTenants`, utilizzata per ottenere la lista dei *my tenant*.

GetMasterTenant

Questa componente ha il compito di restituire il *master tenant* richiesto.
È costituita da:

- *Route API*: `/master/api/tenants/:id/master`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*.

Status	Descrizione	Risposta
200	*Master tenant* recuperato con successo.	Rappresentazione *JSON* dell'oggetto `MasterTenant`.
400	Errore nel recupero del *master tenant* richiesto a causa di un errore nella richiesta.	Richiesta invalida.
404	*Master tenant* richiesto non trovato.	*Master tenant* inesistente.
500	Errore nel recupero del *master tenant* richiesto a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.3: Esiti possibili di GetMasterTenant.

GetMyTenant, GetTenantEfmModule

In modo simile a `GetMasterTenant`, sono state definite le *route* `GetMyTenant` e `GetTenantEfmModule`, utilizzate per ottenere rispettivamente il *my tenant* indicato e il modulo *efm* associato al *tenant* richiesto.

GetTenantIndexes

Questa componente si occupa, dato un *tenant*, di restituire la lista di tutti gli indici, ciascuno associato ad un *flag* che indica l'esistenza rispetto al *tenant* richiesto.
È costituita da:

- *Route API*: `/master/api/tenants/:id/indexes`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*.

Status	Descrizione	Risposta
200	Lista di esistenza degli indici rispetto a un *tenant* recuperata con successo.	Rappresentazione *JSON* dell'oggetto `TenantIndexesExistence`.
400	Errore nel recupero della lista di esistenza degli indici a causa di un errore nella richiesta.	Richiesta invalida.
		Continua nella pagina seguente ...

Tabella 4.4 - Continuo della tabella

Status	Descrizione	Risposta
404	*Tenant* richiesto non trovato	*Tenant* inesistente.
404	Lista degli indici di *default* non trovata	Lista degli indici di *default* inesistente.
500	Errore nel recupero del *my tenant* richiesto a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.4: Esiti possibili di GetTenantIndexes.

NewMasterTenant

Questa componente ha il compito di creare un *tenant* nel *master database*.
È costituita da:

- *Route API*: `/master/api/tenants/master`;
- Metodo: POST;
- *Body* della richiesta: rappresentazione *JSON* di un oggetto `MasterTenant`.

Status	Descrizione	Risposta
200	Creazione del *master tenant* avvenuta con successo.	Rappresentazione *JSON* dell'oggetto `MasterTenant` appena creato.
400	Creazione del *master tenant* fallita a causa di una richiesta non valida.	Richiesta non valida.
500	Creazione del *master tenant* fallita a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.5: Esiti possibili di NewMasterTenant.

NewMyTenant, NewEfmModule

In modo simile a `NewMasterTenant`, sono state definite le *route* `NewMyTenant` e `NewEfmModule`, utilizzate per creare rispettivamente un'entità *my tenant* e il modulo *efm* associato ad un *tenant*.

Questa componente ha il compito di creare il modulo *efm* del *tenant* specificato.
È costituita da:

- *Route API*: `/master/api/tenants/:id/efm-module`;
- Metodo: POST;
- Parametro della richiesta: id del *tenant*;
- *Body* della richiesta: rappresentazione *JSON* di un oggetto `EfmModule`.

Status	Descrizione	Risposta
200	Creazione del modulo *efm* avvenuta con successo.	Rappresentazione *JSON* dell'oggetto `EfmModule` appena creato.
400	Creazione del modulo *efm* fallita a causa di una richiesta non valida.	Richiesta non valida.
500	Creazione del modulo *efm* fallita a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.6: Esiti possibili di NewEfmModule.

UpdateMasterTenant

Questa componente ha il compito di aggiornare il *master tenant* specificato.
È costituita da:

- *Route API*: `/master/api/tenants/:id/master`;
- Metodo: PUT;
- Parametro della richiesta: id del *tenant*;
- *Body* della richiesta: rappresentazione *JSON* di un oggetto `MasterTenant`.

Status	Descrizione	Risposta
200	Aggiornamento del *master tenant* avvenuto con successo.	Rappresentazione *JSON* dell'oggetto `MasterTenant` appena aggiornato.
400	Aggiornamento del *master tenant* fallito a causa di una richiesta non valida.	Richiesta non valida.
404	Aggiornamento del *master tenant* fallito a causa dell'inesistenza del *tenant* associato.	Richiesta non valida.
500	Aggiornamento del *master tenant* fallito a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.7: Esiti possibili di UpdateMasterTenant.

UpdateMyTenant, UpdateEfmModule

In modo simile a `UpdateMasterTenant`, sono state definite le *route* `UpdateMyTenant` e `UpdateEfmModule`, utilizzate per modificare rispettivamente un'entità *my tenant* e il modulo *efm* associato ad un *tenant*.

UpdateTenantIndexes

Questa componente ha il compito di aggiornare la lista di esistenza degli indici rispetto a un *tenant*.

È costituita da:

- *Route API*: `/master/api/tenants/:id/indexes`;
- Metodo: PUT;
- Parametro della richiesta: id del *tenant*;
- *Body* della richiesta: rappresentazione *JSON* di una lista di oggetti `IndexChoice`.

Status	Descrizione	Risposta
200	Aggiornamento della lista di esistenza degli indici rispetto a un *tenant* avvenuto con successo.	Rappresentazione *JSON* dell'oggetto `TenantIndexesExistence` appena aggiornato.
400	Aggiornamento della lista di esistenza degli indici rispetto a un *tenant* fallito a causa di una richiesta non valida.	Richiesta non valida.
404	*Tenant* richiesto non trovato	*Tenant* inesistente.
404	Lista degli indici di default non trovata	Lista degli indici di default inesistente.
500	Aggiornamento della lista di esistenza degli indici rispetto a un *tenant* fallito a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.8: Esiti possibili di UpdateTenantIndexes.

GetTenantUsersStats

Questa componente ha il compito di restituire le statistiche degli utenti rispetto a un *tenant*.

È costituita da:

- *Route API*: `/master/api/tenants/:id/users/stats`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*;
- Parametri URL:
  - `since`: data in formato *Unix* che indica la data di inizio delle statistiche;
  - `till`: data in formato *Unix* che indica la data di fine delle statistiche;
  - `pad`: booleano che indica se formattare le statistiche in modo che i valori non presenti siano sostituiti da 0 nel caso di modalità non cumulativa, o dal numero precedente nel caso di modalità cumulativa;
  - `cum`: booleano che indica se calcolare le statistiche in modo cumulato;

- `res`: stringa che indica la risoluzione temporale delle statistiche ("d" (giornaliera), "m" (mensile), "y" (annuale)).

Status	Descrizione	Risposta
200	Statistiche degli utenti rispetto a un *tenant* recuperate con successo.	Rappresentazione *JSON* dell'oggetto `TenantStats`.
400	Errore nel recupero delle statistiche degli utenti a causa di un errore nella richiesta.	Richiesta invalida.
404	*Tenant* richiesto non trovato	*Tenant* inesistente.
500	Errore nel recupero delle statistiche degli utenti a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.9: Esiti possibili di GetTenantUsersStats.

GetTenantTicketsStats, GetTenantAssetsStats, GetTenantInspectionsStats

In modo simile a `GetTenantUsersStats`, sono state definite le *route* `GetTenantTicketsStats`, `GetTenantAssetsStats` e `GetTenantInspectionsStats`, per calcolare e restituire rispettivamente le statistiche di *tickets*, *assets* e ispezioni.

GetTenantUsersCount

Questa componente ha il compito di restituire il conteggio degli utenti rispetto a un *tenant*. È costituita da:

- *Route API*: `/master/api/tenants/:id/users/count`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*;
- Parametri URL:
  - `since`: data in formato *Unix* che indica la data di inizio delle statistiche;
  - `till`: data in formato *Unix* che indica la data di fine delle statistiche.

Status	Descrizione	Risposta
200	Conteggio degli utenti rispetto a un *tenant* recuperato con successo.	Rappresentazione *JSON* di un valore intero.
400	Errore nel recupero del conteggio degli utenti a causa di un errore nella richiesta.	Richiesta invalida.
404	*Tenant* richiesto non trovato	*Tenant* inesistente.

Continua nella pagina seguente ...

Tabella 4.10 - Continuo della tabella

Status	Descrizione	Risposta
500	Errore nel recupero del conteggio degli utenti a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.10: Esiti possibili di GetTenantUsersCount.

GetTenantTicketsCount, GetTenantAssetsCount, GetTenantInspectionsCount

In modo simile a `GetTenantUsersCount`, sono state definite le *route* `GetTenantTicketsCount`, `GetTenantAssetsCount` e `GetTenantInspectionsCount`, per calcolare e restituire rispettivamente il conteggio di *tickets*, *assets* e ispezioni.

Questa componente ha il compito di restituire il conteggio delle ispezioni rispetto a un *tenant*.

È costituita da:

- *Route API*: `/master/api/tenants/:id/inspections/count`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*;
- Parametri URL:
  - `since`: data in formato *Unix* che indica la data di inizio delle statistiche;
  - `till`: data in formato *Unix* che indica la data di fine delle statistiche.

Status	Descrizione	Risposta
200	Conteggio delle ispezioni rispetto a un *tenant* recuperato con successo.	Rappresentazione *JSON* di un valore intero.
400	Errore nel recupero del conteggio delle ispezioni a causa di un errore nella richiesta.	Richiesta invalida.
404	*Tenant* richiesto non trovato	*Tenant* inesistente.
500	Errore nel recupero del conteggio delle ispezioni a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.11: Esiti possibili di GetTenantInspectionsCount.

GetTenantGeoContainersStats

Questa componente ha il compito di restituire le statistiche dei *geocontainers* rispetto a un *tenant*.

È costituita da:

- *Route API*: `/master/api/tenants/:id/geocontainers/stats`;
- Metodo: GET;
- Parametro della richiesta: id del *tenant*;

- Parametri URL:
  - `since`: data in formato *Unix* che indica la data di inizio delle statistiche;
  - `till`: data in formato *Unix* che indica la data di fine delle statistiche;
  - `type`: lista di stringhe che indica i tipi di *geocontainer* di cui si vogliono conoscere le statistiche;

Status	Descrizione	Risposta
200	Statistiche dei *geocontainers* rispetto a un *tenant* recuperate con successo.	Rappresentazione *JSON* dell'oggetto `TenantStats`.
400	Errore nel recupero delle statistiche dei *geocontainers* a causa di un errore nella richiesta.	Richiesta invalida.
404	*Tenant* richiesto non trovato	*Tenant* inesistente.
500	Errore nel recupero delle statistiche dei *geocontainers* a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.12: Esiti possibili di GetTenantGeoContainersStats.

GetIndexes

Questa componente ha il compito di restituire la lista di tutti gli indici.
È costituita da:

- *Route API*: `/master/api/indexes`;
- Metodo: GET;
- *Body* della richiesta: vuoto.

Status	Descrizione	Risposta
200	Lista degli indici recuperata con successo.	Rappresentazione *JSON* della lista di oggetti `Index`.
404	Lista degli indici non trovata.	Lista degli indici inesistente.
500	Errore nel recupero della lista degli indici a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.13: Esiti possibili di GetIndexes.

GetIndex

Questa componente ha il compito di restituire l'indice richiesto.
È costituita da:

- *Route API*: `/master/api/indexes/:id`;
- Metodo: GET;
- Parametro della richiesta: id dell'indice.

Status	Descrizione	Risposta
200	Indice recuperato con successo.	Rappresentazione *JSON* dell'oggetto `Index`.
400	Errore nel recupero dell'indice richiesto a causa di un errore nella richiesta.	Richiesta invalida.
404	Indice richiesto non trovato.	Indice inesistente.
500	Errore nel recupero dell'indice richiesto a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.14: Esiti possibili di GetIndex.

ApplyIndexToAllTenants

Questa componente ha il compito di applicare l'indice richiesto a tutti i *tenant*.
È costituita da:

- *Route API*: `/master/api/indexes/:id/apply`;
- Metodo: PUT;
- Parametro della richiesta: id dell'indice.

Status	Descrizione	Risposta
200	Applicazione dell'indice a tutti i *tenant* avvenuta con successo.	Rappresentazione *JSON* dell'oggetto `Index`.
400	Applicazione dell'indice a tutti i *tenant* fallita a causa di un errore nella richiesta.	Richiesta invalida.
404	Indice richiesto non trovato.	Indice inesistente.
500	Applicazione dell'indice a tutti i *tenant* fallita a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.15: Esiti possibili di ApplyIndexToAllTenants.

GetUploadSignedURL

Questa componente ha il compito di restituire un *URL* firmato per l'*upload* di un file su un *bucket S3*.
È costituita da:

- *Route API*: `/master/api/signedupload`;
- Metodo: GET;
- *Body* della richiesta: stringa rappresentante il nome del file da caricare.

Status	Descrizione	Risposta
200	URL firmato per l'*upload* di un file su un bucket *S3* recuperato con successo.	Rappresentazione *JSON* dell'url firmato.
500	Errore nel recupero del *URL* firmato per l'*upload* di un file su un bucket *S3* a causa di un errore interno del *server*.	Errore interno al *server*.

Tabella 4.16: Esiti possibili di GetUploadSignedURL.

4.3.3 Sistemi di persistenza

Dovendo interagire con i *database* preesistenti del prodotto *SYN*, i sistemi di persistenza dell'applicazione sviluppata seguono la loro struttura, già definita a priori.

4.3.3.1 Paradigma non relazionale

La piattaforma *SYN* interagisce con grandi quantità e varietà di dati. Per questo motivo ha bisogno di strutture altamente flessibili per rappresentare e memorizzare i dati.

Il paradigma non relazionale, noto anche come Not Only Structured Query Language (NoSQL), rappresenta una modalità di gestione di basi di dati che si discosta dai tradizionali modelli relazionali, basati su tabelle, favorendo modelli di dati più flessibili e scalabili, particolarmente adatti a gestire dati eterogenei e in rapida evoluzione, come quelli trattati da *SYN*.

MongoDB è uno dei *database* non relazionali più diffusi. In *MongoDB*, i dati sono organizzati in documenti Binary JSON (BSON), che consentono di rappresentare strutture dati annidate, in modo simile a quanto accade in un oggetto complesso nella programmazione Object Oriented Programming (OOP)$_G$.

Questa modalità di rappresentazione è molto diversa dal modello relazionale, in cui i dati sono strutturati in tabelle predefinite e rigidamente schematizzate. In un *database* relazionale, le relazioni tra tabelle vengono gestite tramite chiavi primarie e chiavi esterne, il che può rendere complesse le operazioni di lettura e scrittura in presenza di dati molto interconnessi. Al contrario, *MongoDB* memorizza informazioni in documenti autonomi, che possono includere insiemi di valori e documenti annidati, eliminando la necessità di eseguire operazioni di unioni di tabelle complesse e facilitando l'accesso ai dati correlati, aumentando notevolmente le prestazioni delle operazioni di lettura.

L'adozione di un *database* non relazionale nella piattaforma *SYN* ha portato numerosi vantaggi rispetto all'utilizzo di tradizionali *database* relazionali:

- Flessibilità degli schemi di rappresentazione: *MongoDB* consente di aggiungere nuovi campi ai documenti senza la necessità di modificare l'intero schema del *database*. Questo aumenta la flessibilità e permette di adattare il *database* alle nuove esigenze dell'applicazione senza bisogno di una ristrutturazione complessa;

- Prestazioni: *MongoDB* è ottimizzato per operazioni di lettura e scrittura veloci, grazie alla sua architettura basata su documenti. L'assenza di *join* complessi e la possibilità di eseguire query$_G$ direttamente sui documenti permettono di ridurre il tempo di esecuzione delle operazioni;

- Gestione dei dati non strutturati: *MongoDB* eccelle nella gestione di dati non strutturati, tipici delle applicazioni web moderne, come dati *JSON* o *XML*. Questo lo rende una scelta ideale per applicazioni che devono integrare e analizzare grandi varietà di dati eterogenei;

- Supporto per *query* geo-spaziali: *MongoDB* include funzionalità avanzate per la gestione di dati geospaziali, permettendo di eseguire *query* geospaziali come ricerca di punti nello spazio, calcolo di distanze e gestione di coordinate geografiche, facilitando notevolmente le operazioni di mappatura di *SYN*.

4.3.3.2 Configurazione

Nel contesto dell'architettura *multi-tenant* adottata da *Datasoil*, l'utilizzo di *MongoDB* consente di strutturare i dati in modo efficiente e scalabile, mantenendo al contempo una chiara separazione tra i dati dei diversi clienti. L'architettura di *MongoDB* è suddivisa in *cluster*, *database* e collezioni, ognuno dei quali svolge un ruolo fondamentale nella gestione dei dati.

Un *cluster* in *MongoDB* rappresenta un'unità di distribuzione e scalabilità. Un *cluster* può essere composto da uno o più nodi, ognuno dei quali può servire richieste di lettura e scrittura. Nel contesto *multi-tenant* di *Datasoil*, ogni *cluster* è configurato per ospitare più *database*, ognuno dedicato a un singolo cliente.

All'interno di ogni *database*, i dati sono organizzati in collezioni. Una collezione in *MongoDB* è un insieme di documenti, cioè rappresentazioni di dati in formato *BSON*. Le collezioni vengono utilizzate per suddividere i dati di un cliente in diverse categorie, come utenti, *asset* e *ticket*.

In *Datasoil*, ogni cluster *MongoDB* prevede l'utilizzo di un *database master* e un *database* individuale per ogni *tenant*.

4.3.3.3 Master database

Il *database master* svolge un ruolo centrale nella gestione dei *tenant*. Contiene la collezione *tenants*, nella quale ogni documento rappresenta un *tenant* specifico, memorizzando informazioni generali per la loro gestione, tra cui:

- Nome: identificativo testuale del *tenant*;

- *Host*: informazioni sull'*host* del *tenant*;

- Tipo di autenticazione: dettagli sul metodo di autenticazione utilizzato dal *tenant*;

- Informazioni sulla modalità di *login*: configurazioni specifiche relative al *login* del *tenant*, la cui struttura varia a seconda del tipo di autenticazione impostato;

- Informazioni di default: informazioni di default del *tenant*, personalizzabili poi a livello di utente. Tra queste sono compresi il ruolo default degli utenti, il *locale* e il *zone info*.

4.3.3.4 Tenant Database

Ogni *tenant* è poi gestito in un proprio *database* dedicato, il cui nome corrisponde all'ID del documento del *tenant* nella collezione *tenants* del *database* master. Memorizza tutte le informazioni e i dati dell'applicazione *SYN* relativi al *tenant* specifico. In questo *database* è presente una collezione *mytenant*, che contiene informazioni specifiche del *tenant*, diverse da

quelle memorizzate nel *database master*, e molte altre collezioni che organizzano tutti i dati dell'applicazione *SYN* relativi al *tenant*. Nel contesto dell'applicazione sviluppata, sono state utilizzate le collezioni *users*, *tickets*, *assets* e *inspections*, che memorizzano al loro interno le medesime entità.

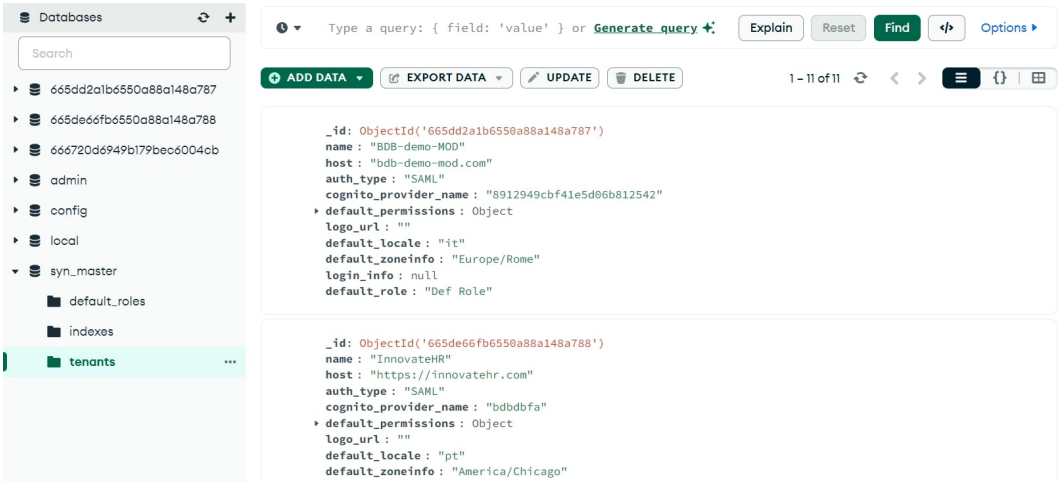

Figura 4.4: Struttura dei sistemi di persistenza visti da *MongoDB Compass*

Capitolo 5

Dettagli implementativi

L'attività di codifica ha lo scopo di realizzare concretamente il prodotto *software* seguendo l'architettura individuata.

5.1 Autenticazione federata tramite JWT

5.1.1 Processo di autenticazione

L'autenticazione federata è un processo che permette agli utenti di un'applicazione di effettuare l'accesso utilizzando le credenziali di un *provider* di identità esterno, come *Google*, *Amazon* o *Facebook*. Questo metodo di autenticazione migliora l'esperienza utente, riducendo il numero di password che gli utenti devono gestire, e aumenta la sicurezza del processo grazie all'utilizzo di *provider* di identità affidabili e riconosciuti.

In questa sezione, verrà esaminato il processo di autenticazione federata implementato, che consente l'accesso all'applicazione sviluppata esclusivamente a un gruppo ristretto di utenti appartenenti al *workspace Google* dell'azienda. Questo processo utilizza tecnologie come JSON Web Token (JWT), Open Authorization 2 (OAuth2) e *Cognito*, con un'implementazione specifica basata sull'approccio Proof Key for Code Exchange (PKCE).

JWT è uno standard aperto utilizzato per creare token accessibili tramite *JSON*, che permettono la trasmissione sicura di informazioni tra due parti. Ogni *JWT* è composto da tre parti:

- *Header*: contiene il tipo di *token* e l'algoritmo di firma;
- *Payload*: contiene le rivendicazioni, ovvero le dichiarazioni utili a verificare la validità e la provenienza del *token*;
- *Signature*: è la firma del *token*, ottenuta combinando l'*header* e il *payload* e firmandoli con un algoritmo di *hashing*.

OAuth2 è un protocollo di autorizzazione standard che permette alle applicazioni di ottenere accesso limitato agli account utente su un servizio *HTTP*, evitando di esporre le credenziali dell'utente ma utilizzando "*token* di accesso" appositi per autenticare le richieste *API*.

Cognito è un servizio di gestione delle identità fornito da *AWS* che consente di aggiungere funzionalità di registrazione, accesso e controllo degli accessi ad applicazioni web. *Cognito* supporta l'autenticazione tramite vari *provider* di identità, come *Google*, *Facebook* e *Apple*, e offre l'integrazione con *SAML* e *OpenID Connect*.

PKCE è un'estensione del flusso di autorizzazione di *OAuth2*, progettata per migliorare la sicurezza delle applicazioni. *PKCE* introduce due nuovi parametri nel flusso di autorizzazione:

- *Code Verifier*: una stringa generata casualmente dall'applicazione;
- *Code Challenge*: una trasformazione crittografica del *code verifier*.

Il processo di *PKCE* prevede i seguenti passaggi:

- Generazione del *code verifier* e *code challenge*: l'applicazione genera un *code verifier* e calcola il relativo *code challenge*;
- Richiesta di autorizzazione: l'applicazione include il *code challenge* nella richiesta di autorizzazione;
- Concessione del codice di autorizzazione: l'utente effettua l'autenticazione e l'applicazione riceve un codice di autorizzazione;
- Scambio del codice di autorizzazione con il *token* di accesso: l'applicazione invia il *code verifier* insieme al codice di autorizzazione per ottenere un *token* di accesso.

Nel contesto dell'applicazione sviluppata, l'autenticazione federata con *Google* tramite *Cognito* segue questo flusso:

- Inizio del flusso di autenticazione: l'utente tenta di accedere all'applicazione;
- Generazione del codice di verifica (*PKCE*): l'applicazione genera un codice di verifica casuale (*code verifier*) e ne calcola un hash (*code challenge*);
- Reindirizzamento verso *Cognito*: l'applicazione reindirizza l'utente a *Cognito*, inserendo nella richiesta anche il *code challenge*;
- Reindirizzamento verso *Google*: *Cognito* reindirizza l'utente alla pagina di autenticazione di *Google*;
- Autenticazione con *Google*: l'utente inserisce le proprie credenziali *Google* e viene autenticato;
- Concessione dei permessi: l'utente concede all'applicazione l'accesso alle proprie informazioni di profilo *Google*;
- Trasmissione del codice di autorizzazione: dopo l'autenticazione e la concessione dei permessi, *Google* invia un codice di autorizzazione a *Cognito*, che esegue una chiamata *API* verso *Google* per ottenere i *token*; Una volta ricevuti i *token*, *Cognito* reindirizza il *client* su un *URL* preimpostato di *callback* dell'applicazione;
- Scambio del codice di autorizzazione: l'applicazione, utilizzando il codice di autorizzazione ricevuto da *Cognito*, invia una richiesta per ottenere i *token*, allegando alla richiesta anche il *code verifier* generato precedentemente;
- Emissione del *token JWT*: *Cognito* verifica il codice di autorizzazione e il *code verifier*, rilasciando poi i *token* richiesti;
- Accesso alle risorse: l'applicazione utilizza il *token* di accesso per accedere alle risorse protette.

5.1.2 AuthService, AuthContext e hook di supporto

AuthService

Il servizio di autenticazione è stato implementato tramite la definizione della classe `AuthService`, che utilizza l'approccio *PKCE* di *OAuth2* sfruttando il servizio offerto da *Cognito*. La classe `AuthService` definisce i processi di *login* e *logout* e le modalità di gestione dei *token*.

I seguenti punti riassumono la classe `AuthService` e i suoi metodi:

- Inizializzazione: nella creazione di un oggetto `AuthService`, vengono impostati i parametri necessari per l'interazione con *Cognito*, tra cui i vari *endpoint* di accesso alle sue funzionalità e il collegamento *URI* in cui deve avvenire il reindirizzamento, cioè la *route* dell'applicazione in cui *Cognito* invierà un codice di autorizzazione in caso di autenticazione corretta;

- Processo di *login*: il metodo `hostedUILogin` gestisce il processo di *login* tramite l'interfaccia ospitata da *Cognito*. Questo metodo genera un `codeVerifier` e uno `state` casuali, che vengono memorizzati nel *local storage* del *browser* e utilizzati da *Cognito* per confrontare il *client* che richiede la concessione di autorizzazione iniziale con il *client* che scambia il codice di autorizzazione in *token*.

```
1     public async hostedUILogin(fromLocation?: string) {
2         const codeVerifier = generateRandomString(40)
3         const state = `${generateRandomString(40)}___${fromLocation ?? ''}`
4         localStorage.setItem('PKCE_state', state)
5         localStorage.setItem(`${AuthService.codeStoreKey}-${state}`, codeVerifier)
6
7         const codeChallenge = await generateCodeChallenge(codeVerifier)
8         const params = new URLSearchParams({
9             response_type: 'code',
10            client_id: this.authCfg.clientId,
11            scope: this.authCfg.scope  '',
12            redirect_uri: this.authCfg.redirectUri,
13            code_challenge: codeChallenge,
14            code_challenge_method: 'S256',
15            state: state,
16        })
17        location.replace(`${this.authCfg.authEndpoint}?${params.toString()}`)
18    }
```

Codice 5.1: Codice della funzione `hostedUILogin`.

- Scambio del codice di autorizzazione: una volta che l'utente ha effettuato l'autenticazione con successo, *Cognito* redirige il *browser* dell'utente all'*URL* indicato in `redirect_uri` con un codice di autorizzazione. Il metodo `fetchTokens` si occupa di scambiare questo codice per i *token* di accesso, ID e *refresh*, effettuando una richiesta `POST` all'*endpoint* specifico dei *token*;

```
1       public fetchTokens(): Promise<BaseAuthState> {
2         const urlParams = new URLSearchParams(window.location.search)
3         const authCode = urlParams.get('code')
4         const state = urlParams.get('state')
5         const codeVerifier
6           = localStorage.getItem(`${AuthService.codeStoreKey}-${state}`)
7
8         if (!authCode) // Error handling
9         if (!codeVerifier) // Error handling
10
11        return this.postWithUrlFormData(this.authCfg.tokenEndpoint, {
12            grant_type: 'authorization_code',
13            code: authCode,
14            scope: this.authCfg.scope  '',
15            client_id: this.authCfg.clientId,
16            redirect_uri: this.authCfg.redirectUri,
17            code_verifier: codeVerifier,
18        })
19      }
```

Codice 5.2: Codice della funzione fetchTokens.

- Verifica e decodifica dei *token*: per garantire che i *token* non siano scaduti, il servizio offre metodi per decodificare e verificare la validità dei *token*, nonché metodi per aggiornarli nel momento in cui la loro scadenza fosse prossima, considerando un *buffer* di 10 minuti.

AuthContext

In una tipica applicazione *React*, la condivisione di dati tra due componenti avviene tramite il passaggio verticale di *props*, equivalenti ai parametri di una funzione nella programmazione procedurale. Può accadere però, come per il servizio di autenticazione, che alcuni dati e funzioni debbano poter essere accessibili in svariate parti dell'applicazione, rendendo scomodo e complicato l'utilizzo di *props*.

In questi casi, è consigliato l'uso di *React Context*, che fornisce un modo per condividere dati attraverso un albero di componenti senza il bisogno di propagare manualmente i *props* ad ogni livello.

Per rendere accessibile il servizio di autenticazione a tutta la componente *frontend* è stato quindi creato il contesto `AuthContext`, che gestisce lo stato di autenticazione dell'utente e fornisce le funzioni necessarie per il *login*, il *logout* e il recupero dei *token* di autenticazione.

Il componente `AuthProvider` è il *core* del contesto di autenticazione. Avvolgendo il componente di più alto livello dell'applicazione, questo componente riesce a condividere i dati e le funzioni contenute in `AuthContext` con tutta l'applicazione.

```
export const AuthProvider = (children: ReactNode) => {
  const authService = new AuthService()

  // Definizione di authState e authInfo
  // Definizione di logIn, logOut e getIdTokenSilently

  async function initAuth() { /*...*/ }
  useEffect(() => { initAuth() }, [])

  return (
    <AuthContext.Provider value={{
      ...authState, ...authInfo,
      logIn, logOut, getIdTokenSilently
    }}>
      {children}
    </AuthContext.Provider>
  )
}
```

Codice 5.3: Codice della componente AuthProvider.

Il frammento di codice riportato illustra una versione semplificata della componente `AuthProvider`. Sfruttando un'istanza di `AuthService` è stato possibile gestire lo stato di autorizzazione e le informazioni relative all'utente, mettendo a disposizione anche metodi per effettuare il *login*, il *logout* e per accedere ai *token*. Infine, l'uso di `useEffect` ha permesso di invocare al primo caricamento del componente la funzione `initAuth`, che si occupa di verificare la presenza dei *token*, eventualmente reindirizzando l'utente alla pagina di *login*.

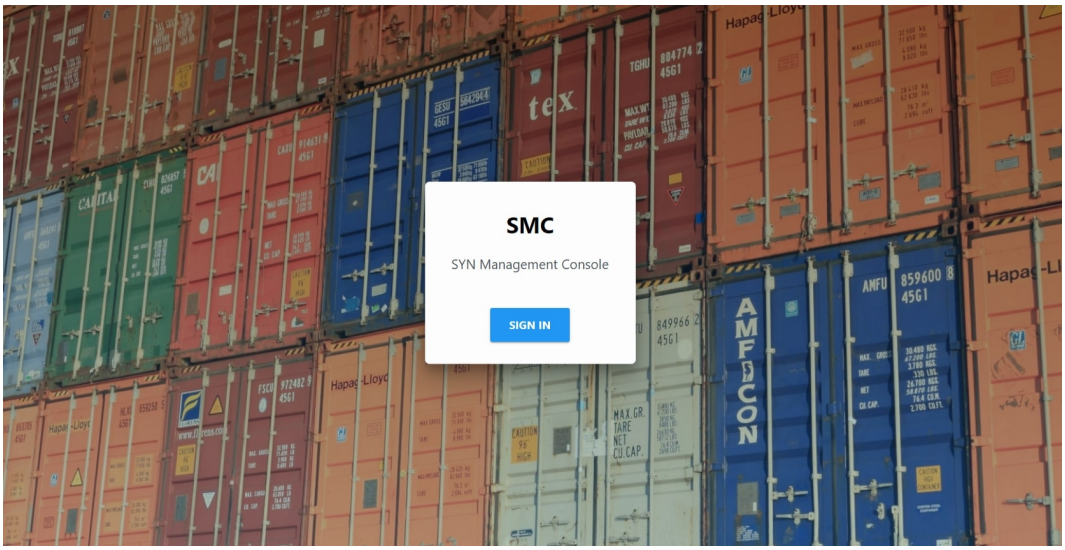

Figura 5.1: Interfaccia grafica della pagina di *login*

Hook di supporto

Per facilitare l'accesso al contesto di autenticazione, sono poi stati implementati i seguenti *hook*:

- `useAuth`: *hook* tramite la quale è possibile accedere direttamente all'oggetto `context` del contesto;
- `useAuthHeader`: *hook* che restituisce il *token* da inserire all'interno di ogni richiesta *API* per poter accedere alle funzionalità offerte dal *backend*;
- `useIsAuthenticated`: *hook* che controlla la validità del *token* e ritorna un valore booleano.

5.1.3 Protezione delle route

Per proteggere l'applicazione, consentendo l'accesso alle varie *route* esclusivamente a utenti autorizzati, è stato implementato il componente `PrivateRoute`, che utilizza `AuthContext` per sapere se l'utente ha effettuato o meno l'accesso. Questo componente agisce da *wrapper* attorno ad ogni *route frontend* dell'applicazione, ad eccezione della pagina di *login*. Se l'utente ha effettuato l'accesso correttamente, il componente `PrivateRoute` ritornerà la *route* richiesta, mentre in caso contrario, l'utente verrà reindirizzato verso la pagina di *login*.

5.1.4 Middleware backend di autenticazione

Per garantire che solo utenti autorizzati possano accedere alle risorse protette del *backend* dell'applicazione, è stato implementato un *middleware* di autenticazione.

Durante la fase di configurazione, viene scaricata la chiave pubblica JSON Web Key (JWK) dal *pool* di utenti di *Cognito* specificato nel file di configurazione. Un *JWK* è un formato *JSON* che rappresenta una chiave crittografica pubblica. In *Cognito*, queste chiavi pubbliche vengono utilizzate per verificare la firma dei *token JWT* emessi dal servizio.

```go
func MiddlewareFunc() gin.HandlerFunc {
  return func(c *gin.Context) {
    tokenStr := c.Request.Header.Get("Authorization")
    if tokenStr == "" {
      c.JSON(http.StatusUnauthorized, "empty auth header")
      return
    }

    tokenStr = strings.TrimPrefix(tokenStr, "Bearer ")
    token, err := mw.ValidateToken(tokenStr)
    if err != nil {
      c.JSON(http.StatusUnauthorized, "invalid auth header")
      return
    }

    c.Set(common.JWTuserKey, token)
    c.Next()
  }
}
```

Codice 5.4: Metodo per generare il *middleware* di autenticazione.

Quando viene ricevuta una richiesta, il *middleware* estrae il *token JWT* dall'intestazione `Authorization` e procede con la sua validazione, suddivisa in due passaggi:

- *Parsing* del *token*: decodifica del *token JWT* utilizzando la chiave pubblica recuperata in fase di configurazione;

- Verifica delle rivendicazioni (*Claims*): una volta decodificato il *token*, il *middleware* procede alla verifica delle rivendicazioni contenute nel *payload* del *token*. Queste includono l'emittente, l'*audience*, il tipo di *token* e la scadenza.

Se l'intestazione è assente o invalida, la richiesta viene respinta con un errore di autorizzazione. In caso contrario, il *middleware* inserisce il *token* valido nel contesto della richiesta e passa il controllo al prossimo *middleware* nella catena.

5.2 Implementazione frontend

5.2.1 Struttura organizzativa

La struttura organizzativa del *frontend* segue le principali convenzioni della libreria *React*. All'interno della cartella `src` sono state raccolte tutte le componenti e le funzioni che formano il *frontend* dell'applicazione web. In particolare, all'interno di questa cartella si trovano le seguenti divisioni:

- `api`: contiene tutti i modelli e le funzioni utilizzate nell'interazione con il *backend*;

- `auth`: modulo che raggruppa tutto ciò che riguarda il processo di autenticazione dell'applicazione web;

- `components`: raccoglie tutti i componenti grafici;

- `routes`: contiene i moduli che si occupano del *routing* dell'applicazione;

- `styles`: raggruppa tutti i file statici e i fogli di stile;

- `utils`: raccoglie tutte le funzioni di utilità rendendole disponibili a tutti i componenti;

- `views`: contiene tutte le pagine dell'applicazione, che organizzano i componenti e gestiscono la loro interazione.

5.2.2 Routing

Per *routing* si intende la capacità di definire e gestire le diverse vie di accesso alle varie componenti dell'applicazione. *React* non include nativamente una soluzione per il *routing*. Per colmare questa mancanza, viene comunemente utilizzata la libreria `react-router-dom`, la quale fornisce strumenti potenti e flessibili per la gestione delle rotte all'interno dell'applicazione. Questa libreria supporta funzionalità avanzate come il *nested routing*, la protezione delle rotte e la gestione dello stato della navigazione, caratteristiche essenziali per applicazioni web complesse e dinamiche.

Il codice presentato di seguito illustra una parte dell'implementazione delle rotte nell'applicazione sviluppata.

```
1  export default function ConsoleRoutes() {
2    return (
3        <Routes>
4          <Route path="/authcbk" element={<LoginView />} />
5          <Route path="/signout"
6                 element={<PrivateRoute loginPath="/authcbk">
7                            <SignoutView />
8                          </PrivateRoute>} />
9          <Route path="/"
10                element={<PrivateRoute loginPath="/authcbk">
11                           <Home />
12                         </PrivateRoute>} />
13         <Route path="/tenants/*"
14                element={<TenantsRoutes />} />
15         <Route path="/indexes/*"
16                element={<IndexesRoutes />} />
17         <Route path="*"
18                element={<PrivateRoute loginPath="/authcbk">
19                           <NoMatch />
20                         </PrivateRoute>} />
21       </Routes>
22   )
23 }
```

Codice 5.5: Routing frontend.

`ConsoleRoutes` definisce una serie di rotte utilizzando la componente `Routes` della libreria di *routing*. Ogni rotta è specificata dalla componente `Route`, che associa ogni percorso a un determinato elemento di *React*. Come accennato precedentemente nella sezione relativa all'autenticazione, ogni *route* dell'applicazione è protetta dalla componente `PrivateRoute`, che condiziona l'accesso alle rotte assicurandosi che solo utenti autenticati possano accedere a determinate viste, reindirizzando in caso contrario l'utente al percorso di *login*.

Le rotte `/tenants/*` e `/indexes/*` utilizzano un *wildcard* (`*`) per indicare che tutte le sottorotte relative ai *tenants* e agli indici sono gestite rispettivamente dai componenti `TenantsRoutes` e `IndexesRoutes`. Questa struttura permette una gestione modulare delle rotte, facilitando la suddivisione del codice in componenti più piccoli e riutilizzabili.

La rotta `*` funge invece da *catch-all*, rispondendo a tutte le richieste non corrispondenti alle rotte precedentemente definite mostrando il componente `NoMatch`, che fornisce un *feedback* all'utente quando questo tenta di accedere a una risorsa inesistente.

5.2.3 Prime React e Prime Flex

Per velocizzare lo sviluppo di nuove interfacce, mantenendo al contempo qualità e consistenza nello stile grafico, *Datasoil* utilizza la libreria *Prime React*, affiancandola alla libreria *CSS Prime Flex*. Entrambe fanno parte della stessa *suite* di componenti, ovvero *PrimeFaces*, sviluppata da *PrimeTek* con lo scopo di fornire strumenti *open source* per facilitare la realizzazione di interfacce utente di applicazioni web.

Per mantenere uniforme lo stile di *SYN* e dell'applicazione del progetto di stage, sono state utilizzate le stesse librerie grafiche, mantenendo invariate anche le loro configurazioni.

Prime React offre una vasta gamma di componenti altamente personalizzabili, responsive e accessibili.

Di seguito viene riportato l'uso della componente `DataTable`, che mostra una tabella offrendo funzionalità integrate di ordinamento, filtraggio e ricerca.

```
1    const [filters, setFilters] = React.useState({
2      'global': { value: null, matchMode: FilterMatchMode.CONTAINS },
3      'auth_type': { value: null, matchMode: FilterMatchMode.IN }
4    });
5
6    <DataTable
7      value={tenants}
8      paginator rows={10} rowsPerPageOptions={[5, 10, 25, 50]}
9      filters={filters} globalFilterFields={["name", "id"]}
10     globalFilter={globalFilter?.trim()} >
11     <Column field="name" header="Name" sortable
12       body={(rowData) => (
13           <Button label={rowData.name} className="p-button-text"
14             onClick={() => navigate(`/tenants/${rowData.id}`)} />
15       )} />
16     <Column field="id" header="ID" sortable />
17     <Column field="host" header="Host" sortable />
18     <Column field="auth_type" header="Auth Type" sortable
19       filter body={(rowData) => <Chip label={rowData.auth_type} />} />
20   </DataTable>
```

Codice 5.6: Esempio di utilizzo di un componente PrimeReact.

Il componente `DataTable` mostra una lista di *tenants* in una tabella paginata, dando all'utente la possibilità di selezionare il numero di righe per pagina, ordinare i dati per proprietà e impostare filtri globali o specifici per colonna. Tramite la proprietà *body* è possibile inoltre definire la rappresentazione e il contenuto di ogni cella, ad esempio utilizzando bottoni, *chip* o altri elementi grafici personalizzati.

L'utilizzo di questi componenti ha reso più efficiente l'attività di implementazione dell'interfaccia grafica, permettendo di dedicare più tempo alla realizzazione delle vere funzionalità dell'applicazione.

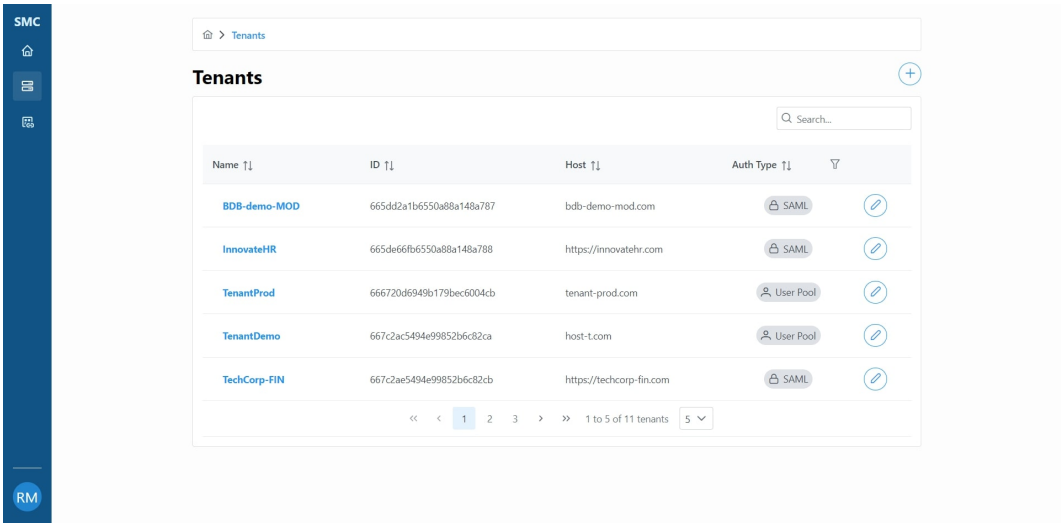

Figura 5.2: Interfaccia grafica del componente `DataTable`

5.2.4 Implementazione form

Essendo l'inserimento e la modifica di informazioni uno degli aspetti principali del progetto, la gestione dei *form* ha rappresentato un aspetto cruciale per garantire una buona esperienza utente nella creazione e nella modifica del *tenant*. Dovendo implementare *form* complessi, è subito risultata necessaria l'introduzione di una libreria esterna in grado di facilitare la gestione di numerosi campi, offrendo funzionalità di validazione e controllo delle relazioni di dipendenze tra campi.

Una delle librerie più utilizzate nel contesto dell'implementazione di *form* è *React Hook Form*, che facilita la gestione dei *form* e migliora le performance evitando l'utilizzo degli stati di *React* per tracciare i cambiamenti dei campi del *form*. L'uso di *TypeScript* con *React Hook Form* consente di specificare rigorosamente i tipi dei dati di ogni campo, aumentando la sicurezza e la robustezza del codice.

L'*API* principale offerta da *React Hook Form* è `useForm`, che restituisce vari metodi e oggetti per la gestione dei *form*, tra cui `register`, `handleSubmit` e `formState`, che permettono rispettivamente di registrare gli input del *form*, gestire l'invio dei dati e monitorare lo stato dei vari campi.

Nello sviluppo di *form* complessi, quando è necessaria la condivisione di stato e funzionalità tra diversi componenti annidati, l'uso del contesto `useFormContext` della libreria diventa particolarmente vantaggioso. Questo contesto consente di accedere alle proprietà di un *form* all'interno di componenti figli, evitando il passaggio manuale di *props* e migliorando la modularità e la riusabilità del codice.

Di seguito viene illustrato un esempio di utilizzo di `useFormContext`.

```
const { register, formState } = useFormContext<MasterTenant>()
return (<>
    <div className="field">
        <label htmlFor="name"
          className={classNames({ 'p-error': !!formState.errors.name })}
        >{'Name'}</label>
        <InputText
            id="name"
            className={classNames({ 'p-invalid': !!formState.errors.name })}
            {...register('name', { required: 'Name is required' })}
            keyfilter={REGEX_ALPHANUM_SIMPLESYMBOLS} />
        {formState.errors.name &&
            <ErrorMessage
                errors={formState.errors}
                name="name"
                render={({ message }) =>
                  <small className="p-error">{message}</small>}
            />}
    </div>
    {/* Implementazione degli altri campi */}
</>)
```

Codice 5.7: Esempio di implementazione di un form.

In questo esempio, il metodo `register` viene utilizzato per collegare l'input del *form* al contesto gestito da *React Hook Form*. L'attributo `formState` si occupa invece di monitorare lo stato corrente del *form*. Viene utilizzato in particolare per gestire gli errori di validazione associati ai campi.

La validazione dei campi del *form* è un aspetto essenziale per garantire l'integrità dei dati inseriti dagli utenti. Tramite il metodo `register`, è possibile definire regole di validazione direttamente nell'attributo dell'input. Nel codice d'esempio, la validazione viene implementata tramite la proprietà `required`, che specifica un messaggio di errore personalizzato nel caso in cui il campo `name` non venga compilato.

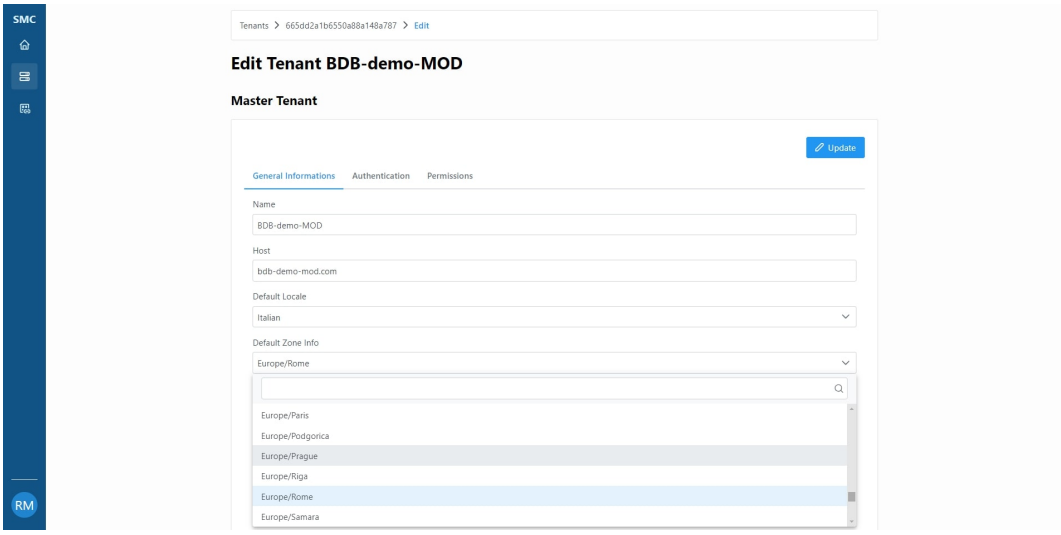

Figura 5.3: Interfaccia grafica del form per la modifica di un *tenant*

5.2.5 Implementazione grafici

Come riportato in (§ 3, Analisi tecnologica), per l'implementazione dei grafici dell'applicazione è stata adottata la libreria *Chart.js*, una libreria *open source* per la visualizzazione di dati, che sfrutta il *rendering* in *canvas* per generare grafici interattivi e personalizzabili.

L'integrazione di *Chart.js* con la libreria grafica *Prime React* ha permesso di semplificare la configurazione iniziale di *Chart.js*, riducendo l'implementazione di un grafico alla sola definizione dei dati da rappresentare.

Di seguito viene presentato un esempio semplificato di utilizzo di *Chart.js* nell'implementazione di un grafico a linea rappresentante l'andamento temporale della creazione di nuovi utenti in un *tenant*.

```
1  const [usersStats, setUsersStats] = useState<TenantStats>();
2  const [chartData, setChartData] = useState({});
3  const [chartOptions, setChartOptions] = useState({});
4
5  const fetchUsersStats = async (params: TenantStatsParams) => { // ... }
6
7  useEffect(() => fetchUsersStats(params), [params]);
8
9  useEffect(() => {
10   const data = {
11     labels: usersStats.timeline.map(t => format(t.t, 'dd/MM/yyyy')),
12     datasets: [{label: 'Users',
13                 data: usersStats.timeline.map(t => t.c) ?? 0}]}
14   const options = {scales: { y: {min: 0, ticks: {precision: 0}}}}
15
16   setChartData(data);
17   setChartOptions(options)
18 }, [usersStats])
19
20 return (
21   {/* ... */}
22   <Chart type="line" data={chartData} options={chartOptions} />
23   {/* ... */}
24 )
```

Codice 5.8: Esempio di implementazione di grafico a linea.

I dati del grafico sono definiti come un oggetto contenente le etichette degli assi (*labels*) e i *dataset*, che specificano i valori da tracciare. Le opzioni del grafico includono invece configurazioni per personalizzare l'aspetto del grafico, per esempio impostando le scale degli assi.

Dal codice sono stati omessi particolari come i parametri della richiesta, che permettono di impostare tramite un *form* il periodo temporale mostrato e alcuni *flag* booleani per l'attivazione del calcolo cumulato e l'inclusione di valori nulli nei *dataset*.

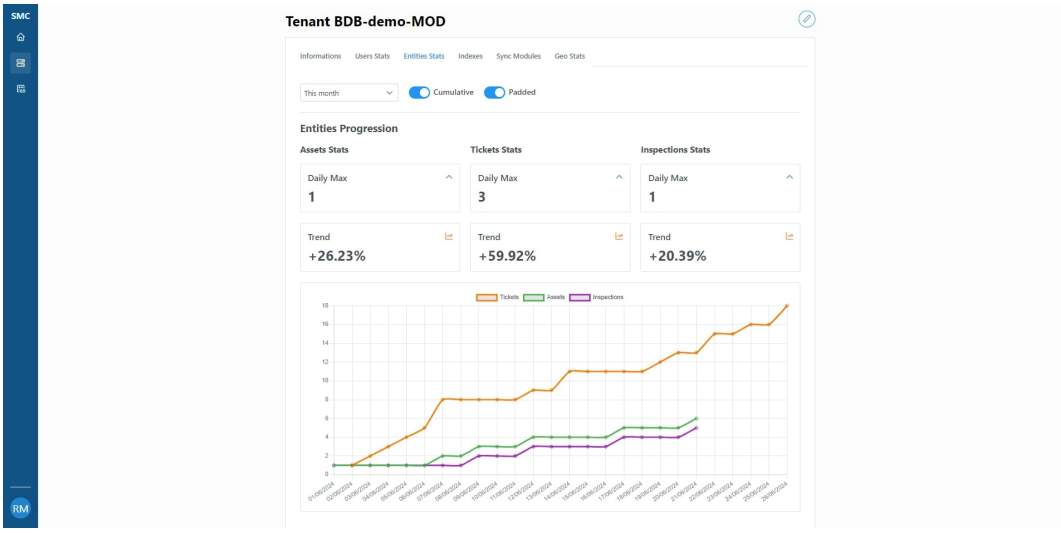

Figura 5.4: Interfaccia grafica della *dashboard* delle *entities*

5.2.6 Interazione con le API del backend

Per semplificare la gestione delle chiamate *API* da parte del *frontend* verso i servizi offerti dal *backend*, sono stati implementati degli *hook* in grado di inviare richieste *HTTP*, gestendo i parametri in entrata e ritornando all'applicazione dati strutturati estratti dalla risposta.

A partire da un *React hook* generico, chiamato `useHttpApi`, che espone metodi capaci di eseguire richieste `POST`, `GET`, `PUT` e `DELETE`, sono stati creati altri *hook* più specifici, che utilizzano i metodi dell'*hook* generico per interfacciarsi con le *API* del *backend*.

Di seguito viene riportato il metodo `apiPost` di `useHttpApi`.

```
1  async function apiPost<T>(api_url: string, data: any) {
2    const token = await getIdTokenSilently()
3    return axios
4      .post<T>(`${ROOT_URL}${api_url}`, data, {
5        headers: { Authorization: `Bearer ${token}` },
6      })
7      .then((response) => response.data)
8      .catch((err: AxiosError) => {
9        if (err.response?.status === 401) {
10         if (logOut) logOut()
11       }
12       throw err
13     })
14 }
```

Codice 5.9: Metodo apiPost di useHttpApi.

Il metodo gestisce la richiesta attraverso `axios`, una libreria che fornisce un *client HTTP* basato sulle `Promise`, cioè oggetti che rappresentano il valore risultante dal completamento o fallimento di un'operazione asincrona. Riceve in input due parametri: `api_url`, che indica il suffisso della *route API* a partire dalla *root*, e `data`, che rappresenta l'oggetto del *body* della richiesta. Dato che ogni *route API* è protetta da un *middleware* di autenticazione, la funzione si occupa anche di aggiungere il *bearer token* di autenticazione nell'*header* `Authorization`.

Tramite l'utilizzo dell'*hook* `useHttpApi` sono stati poi implementati altri tre *hook*, `useTenantsApi` e `useIndexesApi` e `useS3Api`, che definiscono al loro interno tutte le funzioni necessarie per interfacciarsi con ogni *API* fornita dal *backend*.

Di seguito viene riportato l'esempio di `createMasterTenant`, appartenente all'*hook* `useTenantsApi`, che si occupa di effettuare una richiesta `POST` verso l'*API* di creazione di un *master tenant*.

```
1  const createMasterTenant = (tenant: MasterTenant): Promise<MasterTenant> => {
2    return httpApi.apiPost<MasterTenant>('/tenants/master', tenant)
3  }
```

Codice 5.10: Metodo `createMasterTenant` di `useTenantsApi`.

5.3 Implementazione backend

5.3.1 Struttura organizzativa

La componente *backend* sfrutta la struttura standard definita direttamente dalle linee guida del linguaggio *Go*. Ogni progetto in *Go* definisce un modulo, cioè l'unità base di organizzazione del codice.

La struttura classica di un modulo *Go* comprende i seguenti elementi chiave:

- `main package`: è il punto di partenza di ogni applicazione *Go*. Il file principale contiene la funzione `main()`, che è l'*entry point* del programma;
- `go.mod`: specifica il modulo e le sue dipendenze. È simile a un file di configurazione e viene creato utilizzando il comando `go mod init ⟨module_name⟩`. Contiene il nome del modulo e le versioni delle dipendenze richieste;
- `go.sum`: elenca i *checksum* di tutte le dipendenze dichiarate in `go.mod`. Viene aggiornato automaticamente ogni volta che vengono aggiunte, rimosse o aggiornate le dipendenze. Serve a garantire l'integrità delle dipendenze, assicurando che il codice costruito sia riproducibile.

Un modulo può essere formato da un insieme di *package*, ognuno organizzato in un'apposita cartella, che può essere organizzata a sua volta in una struttura gerarchica.

All'interno del modulo del progetto di stage, nominato `github.com/datasoil/syn-master`, sono presenti i seguenti *packages*:

- `aggregationpipes`: contiene tutta la logica riguardante le *aggregation pipelines*, procedure di *MongoDB* che permettono di eseguire operazioni complesse sui dati di un *database*. Verranno approfondite più nel dettaglio in (§ 5, Dettagli implementativi);
- `common`: raccoglie tutte le funzioni di utilità;
- `controllers`: raggruppa la logica del *controller*, le entità *software* responsabili dell'elaborazione dei dati e dell'interazione con i *database*;
- `external`: contiene funzioni e strutture dati per dialogare con servizi esterni, come *Cognito* e *S3*;
- `middlewares`: raggruppa i *middleware*, servizi che permettono di eseguire operazioni o aggiungere informazioni di contesto prima o dopo che la richiesta del *client* venga soddisfatta;

- `models`: contiene tutti i modelli e le interfacce utilizzate dalla componente *backend*;
- `routers`: raggruppa e organizza le routes *API*, cioè definisce i punti e le modalità di accesso verso determinate funzionalità messe a disposizione dal *backend*.

5.3.2 Utilizzo di Gin

In *Gin*, ogni richiesta *HTTP* è rappresentata da un contesto, che raccoglie tutte le informazioni relative alla richiesta e consente di impostare i parametri della risposta. Viene utilizzato per condividere dati tra *middleware* e *handler*, fornendo un'interfaccia coerente per la manipolazione delle richieste e delle risposte.

Dopo l'inizializzazione del *router*, la configurazione di *Gin* prevede la registrazione delle varie *route* dell'applicazione, che permettono di intercettare le richieste in entrata. Ogni *route* è associata a uno o più *handler*, che rappresentano le funzioni eseguite al ricevimento della richiesta.

Questi *handler* possono essere *middleware*, cioè funzioni che gestiscono servizi specifici e che vengono eseguiti prima o dopo la gestione effettiva della richiesta, o *controller*, che si occupano di gestire la logica specifica delle diverse parti dell'applicazione, soddisfacendo le richieste esterne.

Il seguente frammento di codice illustra l'implementazione di `GetMasterTenantController`, il *controller* che gestisce il recupero del *master tenant* richiesto.

```go
func GetMasterTenantController(c *gin.Context) {
  objectId, err := primitive.ObjectIDFromHex(c.Param("id"))
  if err != nil {
    c.JSON(400, gin.H{"error": err.Error()})
    return
  }
  dbClient := c.MustGet(common.DBcli).(*mongo.Client)
  db := dbClient.Database(viper.GetString("db.masterName"))
  tenants := db.Collection(models.MasterTenant{}.DBCollectionName())
  tenant := models.MasterTenant{}
  err = tenants.FindOne(c, bson.M{"_id": objectId}).Decode(&tenant)
  if err != nil {
    if err == mongo.ErrNoDocuments {
      c.JSON(404, gin.H{"error": "tenant not found"})
      return
    }
    c.JSON(500, gin.H{"error": err.Error()})
    return
  }
  c.JSON(200, tenant)
}
```

Codice 5.11: Implementazione di `GetMasterTenantController`.

Dopo un iniziale recupero dell'id del *tenant* dal parametro *URL* della richiesta e un controllo di validità, viene eseguita una richiesta `FindOne` sulla collezione dei *master tenant*, impostando un filtro corrispondente all'id ricercato. Vengono poi gestiti i possibili esiti, ritornando come risposta il *tenant* richiesto o, in caso di fallimento, l'errore opportuno.

5.3.3 Procedura di binding

La procedura di *binding* in *Gin* è un aspetto fondamentale per la gestione delle richieste *API*, in quanto consente di mappare automaticamente i dati in ingresso ai campi delle strutture *Go*, evitando di dover implementare manualmente procedure per la conversione di dati tra diverse rappresentazioni. Questo processo è facilitato dall'uso di *tag JSON* e dal supporto per la validazione integrata, offerti dal popolare *package* `go-playground/validator`.

Quando viene ricevuta una richiesta, *Gin* utilizza i *tag JSON* definiti nelle strutture per eseguire il *binding* dei dati contenuti nel *body* della richiesta verso i campi corrispondenti delle strutture in *Go*.

Di seguito viene riportato l'utilizzo di alcuni *tag* nella struttura `MasterTenant`. I *tag JSON* indicano i nomi dei campi attesi nella richiesta, mentre le direttive *binding* specificano i requisiti di validazione.

```go
type MasterTenant struct {
  ID  primitive.ObjectID
    `json:"id,omitempty" binding:"omitempty,mongodb_id"`
  AuthType  string
    `json:"auth_type" binding:"oneof=SAML USER_POOL,required_without=ID"`
  LoginInfo *LoginInfo
    `json:"login_info" binding:"excluded_if=AuthType SAML,
                                required_if=AuthType USER_POOL"`
  // Ulteriori dichiarazioni
}

func NewMasterTenantController(c *gin.Context) {
  masterTenantRequest := models.MasterTenant{}
  if err := c.ShouldBindJSON(&masterTenantRequest); err != nil {
    c.JSON(400, gin.H{"error": err.Error()})
    return
  }
  // Gestione della richiesta
}
```

Codice 5.12: Utilizzo dei *tag* in `MasterTenant`.

È possibile notare come il *tag binding* sia in grado di eseguire controlli anche piuttosto complessi, che in molti altri *web framework* avrebbero richiesto funzioni di validazione dedicate.

L'avvio del processo di *binding* risiede invece in `NewMasterTenantController`, dove viene chiamata la funzione `ShouldBindJSON`: se l'operazione ha esito positivo, il *tenant* contenuto nella richiesta viene convertito e memorizzato in una struttura apposita, altrimenti viene ritornato un errore di richiesta invalida.

5.3.4 Gestione degli errori

All'interno dell'applicazione vengono frequentemente eseguite operazioni complesse, spesso interagendo con sistemi esterni. Per questo motivo, è stato necessario implementare una gestione degli errori robusta e informativa.

Paradigma di gestione esplicita degli errori in Go

Nel linguaggio *Go*, la gestione degli errori è condotta tramite un paradigma esplicito che richiede la verifica degli errori dopo ogni operazione potenzialmente fallibile. Ogni funzione che potrebbe generare un errore restituisce infatti un secondo valore di tipo `error`, obbligando il programmatore a considerare gli errori in ogni fase del processo, migliorando la robustezza e la manutenzione del codice.

Errori per operazioni complesse

Quando è necessario interagire con più *database*, la mancanza di costrutti nativi, sia nel linguaggio *Go* che in *MongoDB*, per garantire l'atomicità delle operazioni, richiede una gestione accurata dei vari flussi di esecuzione, implicando la necessità di restituire risposte più articolate, capaci di contenere diversi tipi di esiti. Dato che ogni operazione può fallire in modi differenti, è necessario restituire un esito dettagliato e informativo per ogni operazione eseguita. Per affrontare questa complessità, è stata sviluppata una struttura generica in grado di rappresentare i risultati di queste operazioni multiple.

```go
type OperationResult[T any] struct {
  ID      T      `json:"id"`
  Result bool   `json:"result"`
  Message string `json:"message"`
}

type GroupOperationResult[T any] struct {
  ID      T                    `json:"id"`
  Results []OperationResult[T] `json:"results"`
}
```

Codice 5.13: Gestione dei risultati multipli.

La struttura presentata offre numerosi vantaggi. Prima di tutto, grazie all'adozione del tipo generico (`T any`), la struttura è completamente modulare, cioè indipendente da qualsiasi tipo di dato specifico. Questo permette di utilizzare la stessa struttura per una varietà di operazioni differenti, riducendo la necessità di creare nuove strutture per ogni tipo di operazione.

Inoltre, la struttura è altamente informativa. Ogni `OperationResult` contiene un identificativo, un esito booleano e un messaggio descrittivo, fornendo un contesto completo sul risultato di ogni singola operazione. La struttura `GroupOperationResult` permette di raggruppare più `OperationResult`, rendendo possibile la gestione di risultati complessi in un formato organizzato e facilmente comprensibile.

Questa approccio consente di gestire gli errori in modo sistematico e coerente, migliorando la capacità di diagnosticare e risolvere i problemi nelle operazioni complesse e nelle interazioni con più *database*.

5.4 Interazione con Mongo DB

5.4.1 Middleware

L'interazione con diversi *database* all'interno dell'applicazione ha fin da subito reso necessaria una gestione organizzata della connessione al *cluster MongoDB*.

È stato quindi predisposto un *middleware* in grado di inserire all'interno del contesto della richiesta un riferimento al *client Mongo* costruito durante la fase di *setup* delle *route API* all'avvio dell'applicazione.

Questo *middleware* evita la creazione di una variabile globale, che andrebbe contro i principi *REST*, permettendo comunque ad ogni richiesta di accedere a qualsiasi *database* del *cluster* a cui il *client* è connesso.

Il codice sottostante illustra il metodo `NewMongoMiddlewareHandler`, che si occupa di inizializzare la connessione al *cluster* di *Mongo* e creare il gestore del *middleware*, e il metodo `MiddlewareFunc`, che restituisce il *middleware* vero e proprio, registrato poi nella catena di gestione delle richieste.

```go
func NewMongoMiddlewareHandler(mongoUri string) (*MongoMiddlewareHandler, error) {
  c := context.Background()
  client, err := mongo.Connect(c, options.Client().ApplyURI(mongoUri))
  if err != nil {
    return nil, err
  }
  mongoMiddlewareHandler := &MongoMiddlewareHandler{client: client}
  return mongoMiddlewareHandler, nil
}

func (mw *MongoMiddlewareHandler) Disconnect() {
  mw.client.Disconnect(context.Background())
}

func (mw *MongoMiddlewareHandler) MiddlewareFunc() gin.HandlerFunc {
  return func(c *gin.Context) {
    c.Set(common.DBcli, mw.client)
    c.Next()
  }
}
```

Codice 5.14: Middleware di MongoDB.

5.4.2 Tags e risultati delle operazioni di MongoDB

Come spiegato in (§ 5.3.3, Procedura di binding), tramite l'utilizzo dei *tags* è possibile creare delle associazioni tra i campi di una struttura e i diversi formati tramite la quale questa può essere rappresentata.

Questo strumento può essere anche utilizzato per conversioni in *BSON*, un formato binario in grado di rappresentare strutture dati semplici e array associativi, utilizzato in *MongoDB* per il trasferimento e la rappresentazione di documenti.

Nel momento in cui si effettua una qualsiasi operazione con un *database Mongo*, le entità coinvolte vengono prima convertite in *BSON*.

Per esempio, durante un'operazione di scrittura, come uno qualsiasi dei vari tipi di *insert* e *update*, l'istanza della *struct* passata viene trasformata in *BSON*. In questa trasformazione, ogni campo viene rinominato con il nome assegnatogli nel relativo *tag*, e il relativo valore viene convertito in uno dei tipi base di *BSON* o in un *array* associativo in caso di campo di tipo *struct*.

D'altra parte, in un'operazione di lettura, come una ricerca tramite il metodo `Find`, i documenti verranno ritornati in formato *BSON*, e tramite i *tags* è possibile effettuare delle conversioni dirette nella rispettiva struttura in *Go*, sfruttando il *binding* automatico.

I tipi di ritorno delle funzioni di *MognoDB* invece variano a seconda del tipo di operazione, ma si possono raggruppare in due categorie principali:

- Operazioni singole, come `InsertOne`, `UpdateOne` e `FindOne`: queste ritornano una *struct* che rappresenta il risultato della singola operazione. Contengono informazioni utili in base alla richiesta, come nel caso dell'inserimento di un nuovo documento, in cui è presente l'ID del documento creato;

- Operazioni multiple, come `InsertMany`, `UpdateMany` e `Find`: queste ritornano un cursore, un costrutto basato sul pattern *Iterator*. Scorrendolo, è possibile verificare l'esito dell'operazione o decodificare i documenti ritornati in una *struct* opportuna.

5.4.3 Indici su MongoDB

Un indice di un *database* è una struttura dati realizzata per migliorare i tempi di ricerca dei dati. In *MongoDB*, gli indici vengono creati usando *BTree*, una struttura dati ad albero in grado di auto-bilanciarsi mantenendo i dati ordinati e permettendo operazioni di ricerca, accesso sequenziale, inserimenti ed eliminazioni in tempo logaritmico.

Senza la presenza di indici, *MongoDB* dovrebbe scannerizzare ogni documento in una collezione prima di restituire i risultati di una *query*. Nonostante l'evidente miglioramento di prestazioni nell'esecuzione delle *query*, l'aggiunta di un indice causa un rallentamento nelle operazioni di scrittura, poiché ognuna di queste deve essere riflessa in tutti gli indici interessati alla collezione.

In *MongoDB*, un indice è formato da una o più chiavi, che indicano i campi inclusi e il loro ordinamento, e da ulteriori opzioni, che memorizzano informazioni e pongono vincoli sui dati. Tra quest'ultimi, i più usati sono `unique`, che impone il vincolo di unicità, e `sparse`, che impone l'esistenza dei campi specificati nelle chiavi.

In *SYN*, gli indici vengono utilizzati regolarmente per migliorare le prestazioni delle *query*. Dato il grande flusso di dati in lettura ma anche in scrittura, è necessario analizzare approfonditamente le varie soluzioni, valutando con attenzione le conseguenze dell'introduzione di un indice all'interno di una collezione. Questa analisi deve inoltre includere considerazioni specifiche riguardanti i clienti e le loro necessità. A seconda del tipo di utilizzo del prodotto, alcuni indici potrebbero essere o non essere vantaggiosi.

Con lo scopo di rendere più efficienti i processi di analisi e selezione degli indici per ogni *tenant*, è stato individuato un insieme completo, includendo tutti gli indici ad alto impatto, implementando poi una componente dell'applicazione dedicata alla selezione specifica per ogni *tenant*.

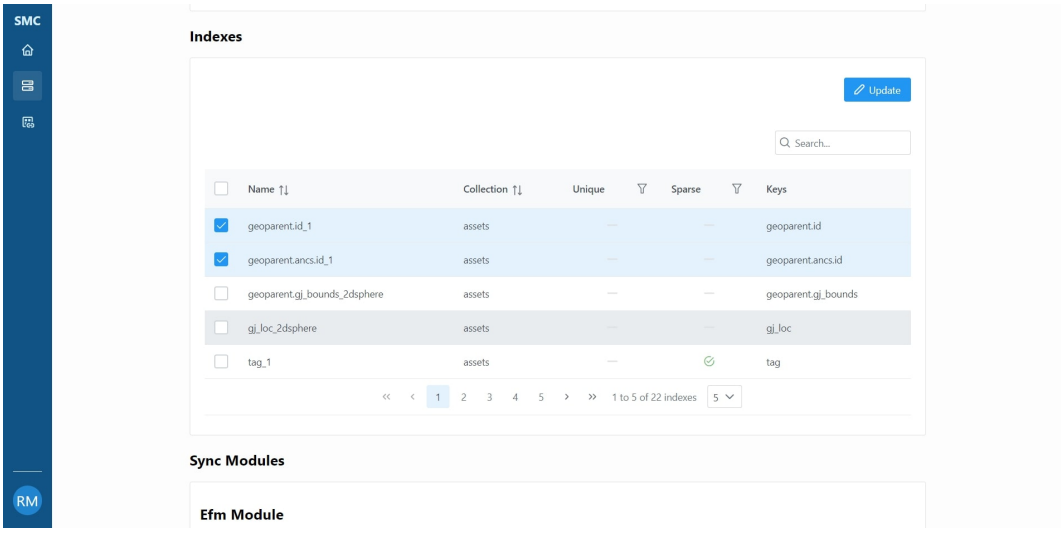

Figura 5.5: Interfaccia grafica della selezione degli indici

5.4.4 Transazioni

Per poter soddisfare alcuni requisiti, l'applicazione ha bisogno di eseguire compiti complessi composti da più operazioni di scrittura o lettura in un *database*.

La maggior parte delle volte, l'esecuzione di queste operazioni complesse deve essere atomica e controllata, in modo che il fallimento di una delle sotto-operazioni non causi perdita di coerenza e integrità dei dati.

Per garantire ciò, è necessario l'uso delle transazioni, che consentono la definizione di unità atomiche di lavoro, la cui esecuzione ha effetto solamente nel caso in cui tutte le operazioni che la compongono terminano con esito positivo.

Nell'applicazione prodotta, l'utilizzo di transazioni ha semplificato notevolmente la gestione delle operazioni multiple. Il seguente frammento di codice riporta un esempio di transazione.

```
1     wc := writeconcern.Majority()
2     txnOptions := options.Transaction().SetWriteConcern(wc)
3
4     session, err := dbClient.StartSession()
5     if err != nil {
6       c.JSON(500, gin.H{"error": err.Error()})
7       return
8     }
9     defer session.EndSession(c)
10
11    _, err = session.WithTransaction(c,
12      func(c mongo.SessionContext) (interface{}, error) {
13        if _, err = myTenantColl.InsertOne(c, myTenantRequest); err != nil {
14          return nil, err
15        }
16
17        if len(rolesListInterface) > 0 {
18          if _, err = myRolesColl.InsertMany(c, rolesListInterface); err != nil {
19            return nil, err
20          }
21        }
22
23        return nil, nil
24      }, txnOptions)
25
26    if err != nil { // ... }
```

Codice 5.15: Esempio di utilizzo di una transazione.

All'interno del metodo `WithTransaction` vengono eseguite le operazioni di scrittura, rispettivamente l'inserimento di un `MyTenant` e l'inserimento della lista di ruoli di default degli utenti.

È possibile inoltre notare l'utilizzo di `defer`, un costrutto in Go che ritarda l'esecuzione di uno *statement* al termine dell'esecuzione della funzione circostante. In questo caso è stato utilizzato per chiudere la sessione con il *database*.

5.4.5 Aggregation pipelines

In *MongoDB*, il processo di definizione ed esecuzione di *query* complesse è facilitato dal concetto di *aggregation pipeline*. Un'*aggregation pipeline* permette di definire un flusso di operazioni che elaborano una serie di documenti e restituiscono un unico risultato finale.

Ogni *aggregation pipeline* può essere formata da più *stage*. Ogni *stage* riceve in input un insieme di documenti dallo *stage* precedente e fornisce il risultato della sua elaborazione allo *stage* successivo.

Raggruppando le operazioni eseguibili all'interno di uno *stage* è possibile comporre *query* molto complesse, non ottenibili tramite la definizione di semplici filtri applicabili alla classica operazione `Find`.

Nel contesto dell'applicazione sviluppata, le *aggregation pipeline* sono state fondamentali nel calcolo delle statistiche per l'implementazione delle *dashboard*.

Il seguente schema illustra le trasformazioni dei documenti che hanno permesso di ottenere una serie temporale in cui, ad ogni intervallo, corrisponde il numero di documenti esistenti fino a quel momento.

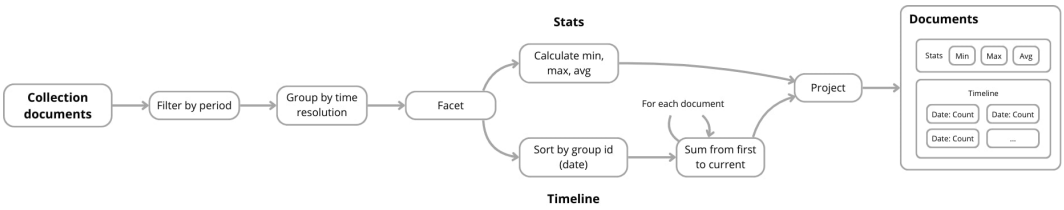

Figura 5.6: Schema di funzionamento dell'*aggregation pipeline* che calcola la progressione

L'intera definizione della *pipeline* è eccessivamente verbosa per essere riportata in questo documento, quindi di seguito viene riportata esclusivamente la definizione dello *stage* che ha permesso di calcolare la progressione temporale. Essendo basata solo sul numero di documenti di una collezione, quindi omettendo qualsiasi particolare riguardante il dato contenuto, questa *pipeline* è stata riutilizzata per produrre le serie temporali di utenti, *tickets*, *assets* e *inspections*. Per semplicità di lettura, il codice omette dettagli come la risoluzione temporale, che permette di utilizzare la stessa *pipeline* per produrre serie temporali di diversa densità, dividendo gli intervalli in periodi annuali, mensili o giornalieri.

```
"timeline": []bson.M{
  {"$group": bson.M{
    "_id": bson.M{
      "year":  bson.M{"$year": "$date_from_id"},
      "month": bson.M{"$month": "$date_from_id"},
      "day":   bson.M{"$dayOfMonth": "$date_from_id"},
    },
    "count": bson.M{"$sum": 1}}
  },
  {"$setWindowFields": bson.M{
    "partitionBy": nil,
    "sortBy": bson.D{{"_id", 1}},
    "output": bson.M{
      "cumulativeCount": bson.M{
        "$sum": "$count",
        "window": bson.M{"documents": bson.A{"unbounded", "current"}}
      }
    }
  }}
}
```

Codice 5.16: Esempio di *stage* di una *aggregation pipeline*.

Lo *stage* timeline è composto da:

- `group`: raggruppa i documenti per giorno di creazione e aggiunge un campo `count` che rappresenta il numero di documenti per ogni gruppo;
- `setWindowFields`: esegue operazioni in uno specifico intervallo di documenti, detto `window`. Contiene:

- `partitionBy`: partiziona i documenti dato un certo campo. In questo caso il valore `nil` evita il partizionamento;
- `sortBy`: ordine i gruppi per il campo `_id`, che nei risultati elaborati da `group` corrisponde alla data per cui i documenti sono stati raggruppati;
- `output`: specifica i *field* che devono essere aggiunti ai documenti ritornati dallo *stage* `setWindowFields`. Contiene il campo `cumulativeCount`, che viene calcolato eseguendo la somma dei campi `count` nella finestra di documenti nell'intervallo compreso tra `unbounded` e `current`, cioè sommando tutti i campi `count` precedenti all'attuale compreso.

5.5 Documentazione

5.5.1 Swagger

Uno degli strumenti più utilizzati per la documentazione delle *API* è *Swagger*. Permette di descrivere direttamente nel codice le *API* in modo strutturato, fornendo un'interfaccia interattiva che migliora la comprensione e l'uso delle *API*.

Swagger è disponibile in *Go* tramite la libreria `swaggo/swag`. Attraverso l'esecuzione di comandi specifici, questa libreria analizza i commenti del codice sorgente e genera automaticamente la documentazione trasformando i commenti in file *JSON*. Questi file sono poi utilizzabili da *Swagger UI* per ottenere una visualizzazione interattiva.

5.5.2 Parametri descrittivi

La descrizione delle *API* in *Swagger* avviene tramite l'annotazione di parametri descrittivi. Di seguito vengono riportati i parametri più utilizzati:

- `@Summary` e `@Description` forniscono un riassunto e una descrizione dettagliata dell'*endpoint*;
- `@ID` assegna un identificatore unico all'*endpoint*;
- `@Tags` classifica l'*endpoint* sotto specifiche categorie;
- `@Router` specifica l'*URL* dell'*endpoint* e il metodo *HTTP* associato;
- `@Security` indica il tipo di autenticazione richiesta;
- `@Accept` e `@Produce` definiscono i formati di input e output accettati e prodotti dall'*endpoint*;
- `@Param` descrive i parametri dell'*endpoint*, compresi il nome, il tipo, la posizione all'interno della richiesta e un *flag* booleano che indica se sono obbligatori;
- `@Success` e `@Failure` documentano i possibili esiti delle richieste, specificando i codici di stato e i tipi di dati restituiti, oltre ai possibili messaggi di errore.

5.5.3 Processo di generazione

Il processo di generazione della documentazione *Swagger* riguardo le *API* prodotte viene effettuato attraverso l'esecuzione di uno *script*. Questo può essere eseguito manualmente o in modo automatico, come parte della pipeline di CI/CD$_G$ definita tramite *CircleCI*.

```
% #!/bin/sh
% swag init --parseDependency -g syn-master.go --generatedTime
% cp docs/swagger.json swaggerui/
```

Codice 5.17: Script di generazione della documentazione.

Questo *script* genera la documentazione *Swagger* analizzando il file `syn-master.go`, cioè l'entrata dell'applicazione *backend*, copiando poi il file *JSON* prodotto nella *directory* `/swaggerui`.

All'interno della *suite* di *Swagger*, *Swagger UI* è lo strumento che consente di visualizzare e interagire con la documentazione prodotta. Utilizza il file *JSON* prodotto dal processo di generazione per creare un'interfaccia web interattiva dove gli sviluppatori possono esplorare le *API*, testare le chiamate e visualizzare il contenuto delle risposte in tempo reale.

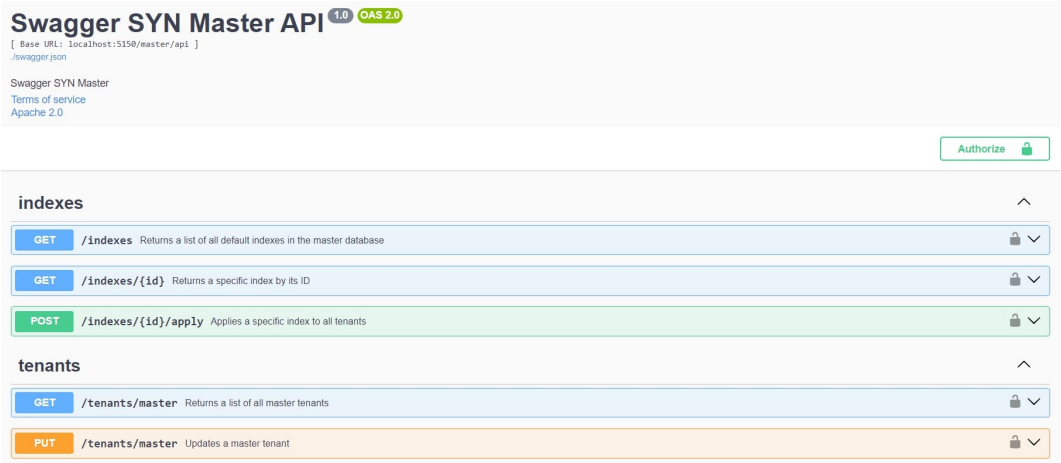

Figura 5.7: Interfaccia grafica del sito web della documentazione

5.5.4 Esposizione della documentazione

Per rendere l'interfaccia di *Swagger UI* accessibile tramite l'applicazione sviluppata, è stato utilizzato il *flag* booleano `-swag`, configurabile all'avvio della componente *backend*.

```go
swag := flag.Bool("swag", false, "Used for activating swagger.")
// ...
if *swag {
  r_m.Static("/swagger", "./swaggerui")
}
```

Codice 5.18: Attivazione della *route* di Swagger UI.

Questo processo permette di accedere alla documentazione attraverso l'*URL* `/swagger`, fornendo un punto centrale per la consultazione e l'interazione con le *API* documentate.

Capitolo 6

Verifica e validazione

Il processo di verifica ha lo scopo di determinare se ciò che viene prodotto è conforme ai requisiti e ai vincoli qualitativi stabiliti. L'obiettivo principale della verifica è garantire la completezza e la correttezza del prodotto, fornendo evidenze oggettive e misurabili. Il processo di validazione, d'altro canto, è finalizzato a determinare se i requisiti stabiliti da contratto siano stati soddisfatti dal prodotto *software*.

6.1 Testing

6.1.1 Test di unità

I test di unità sono essenziali per validare le funzioni di elaborazione all'interno dell'applicazione, come funzioni per la validazione dei parametri in ingresso e funzioni di elaborazione sui dati ricevuti dal *database*. In questo tipo di test vengono utilizzati *mock* di *MongoDB* per simulare il comportamento del *database*, consentendo di testare le funzioni in modo isolato, escludendo ogni dipendenza esterna.

Nella codifica dei test di unità è stata posta particolare attenzione alla validazione delle richieste di creazione e modifica, che costituiscono le funzionalità più importanti e critiche dell'applicazione. Il seguente codice mostra uno dei test effettuati per verificare la correttezza della funzione di validazione di un *master tenant*. In particolare, viene effettuata una richiesta di creazione di un *tenant* con un nome vuoto, assicurandosi che la funzione di validazione generi un errore appropriato.

```go
func TestValidateMasterTenantCreateRequestInvalid_EmptyName(t *testing.T) {
  r := require.New(t)

  // create invalid tenant
  tenant := models.MasterTenant{
    Host:                "testmaster.com",
    AuthType:            models.SAML,
    CognitoProviderName: "cognitoprovidername",
    DefaultLocale:       "en",
    DefaultZoneinfo:     "UTC",
  }

  // validate
  err := validateMasterTenantCreateRequest(&tenant)
  r.NotNil(err)
}
```

Codice 6.1: Esempio di test di unità.

In questo test viene creato un *tenant master* con un nome vuoto e viene chiamata la funzione di validazione `validateMasterTenantCreateRequest`, asserendo tramite la libreria `require` di `github.com/stretchr/testify` che l'errore ritornato non sia nullo.

6.1.2 Test di integrazione

I test di integrazione verificano il corretto funzionamento delle varie componenti dell'applicazione nel loro insieme, testando le rotte e i *controller* con parametri specifici per assicurare il corretto comportamento dell'applicazione a fronte di specifiche condizioni.

Ogni test di integrazione realizzato imposta inizialmente l'ambiente di esecuzione, simulando uno stato specifico dell'applicazione, ed effettua chiamate verso le rotte in analisi, verificando l'uguaglianza tra il comportamento effettivo e quello aspettato.

Per questo tipo di test è stato utilizzato un *database* apposito, che viene svuotato ad ogni esecuzione per garantire che i test siano ripetibili e non influenzati da dati residui.

Il seguente esempio mostra uno dei test di integrazione implementati per verificare il corretto funzionamento della *route* che si occupa di restituire al richiedente la lista completa dei *master tenant*.

```go
func TestGetMasterTenants_Pass(t *testing.T) {
  r := require.New(t)
  clearColls(r)
  // create tenants
  tenant1 := models.MasterTenant{ // ... }
  tenant2 := models.MasterTenant{ // ... }

  // directly insert tenants
  res1, err := getMasterDB().Collection(models.MasterTenant{}.DBCollectionName())
    .InsertOne(context.Background(), tenant1)
  r.NoError(err)
  tenant1.ID = res1.InsertedID.(primitive.ObjectID)
  res2, err := getMasterDB().Collection(models.MasterTenant{}.DBCollectionName())
    .InsertOne(context.Background(), tenant2)
  r.NoError(err)
  tenant2.ID = res2.InsertedID.(primitive.ObjectID)

  // get tenants
  resTenants := []models.MasterTenant{}
  err = unmarshalObject(getObject(t, "/tenants/master", nil), &resTenants)

  r.NoError(err)
  r.Len(resTenants, 2)
  r.Contains(resTenants, tenant1)
  r.Contains(resTenants, tenant2)
}
```

Codice 6.2: Esempio di test di integrazione.

Dopo la chiamata a `clearColls`, che svuota le collezioni interessate, vengono creati e inseriti direttamente nel *database* due *master tenant*. Viene poi chiamata la *route* `/tenants/master` per ottenere la lista dei *master tenant*, e i risultati vengono deserializzati in una *slice* di `MasterTenant`. Infine, il test verifica che i *tenant* ottenuti siano esattamente quelli inseriti.

Complessivamente, è stato raggiunto un *code coverage* dell'`85%`. È importante notare che questa percentuale non include tutti gli *statement* di gestione degli errori del *database*, poiché in questo tipo di test non è possibile simulare risposte di fallimento di esecuzione o di connessione.

6.2 Validazione

Essendo lo strumento sviluppato finalizzato a un uso interno, durante lo sviluppo è stato possibile effettuare degli incontri periodici di validazione con i futuri utilizzatori, ottenendo preziosi *feedback* riguardanti la correttezza delle funzionalità sviluppate e la qualità dell'interfaccia utente proposta.

Grazie a queste riunioni è stato possibile verificare la soddisfazione dei requisiti e ottenere una stima più precisa dello stato di avanzamento del prodotto.

Capitolo 7

Integrazione e distribuzione continue

Questo capitolo ha il compito di illustrare gli strumenti utilizzati e le procedure messe in atto per rendere possibile una corretta applicazione dei concetti di integrazione continua (*CI*) e distribuzione continua (*CD*).

Lo scopo dei processi *CI/CD* è semplificare e accelerare il ciclo di sviluppo del prodotto *software*, favorendo un approccio più incrementale e controllato.

L'integrazione continua consiste nell'integrazione frequente e automatica delle modifiche in un *repository* condiviso del codice sorgente.

La distribuzione continua è un processo durante il quale le modifiche apportate al codice vengono integrate, testate e distribuite. La distribuzione continua automatizza il rilascio su un *repository* del codice convalidato, in seguito all'automazione delle *build* e dei test di unità e integrazione nella fase di *CI*. Al termine di questo processo, il *team* operativo è in grado di distribuire rapidamente l'app in produzione. La distribuzione continua infatti non si occupa anche della fase di rilascio, ma garantisce esclusivamente che il codice dell'applicazione venga mantenuto costantemente in uno stato rilasciabile. In *Datasoil*, il processo di rilascio è volontariamente manuale per motivi legati alla sicurezza e per permettere agli operatori di avere un controllo maggiore sul prodotto finale.

7.1 Descrizione degli strumenti di CI/CD

In questa sezione vengono riportati gli strumenti utilizzati nelle attività di integrazione e rilascio del prodotto. Per mantenere coerenti le attività citate tra i diversi prodotti dell'azienda, è stata presa la decisione di utilizzare gli stessi strumenti, configurando le relative procedure in modo simile, facilitando il monitoraggio e la gestione delle risorse.

7.1.1 Git e GitHub

Il versionamento del codice sorgente è stato gestito tramite *Git*, con l'ausilio della piattaforma *GitHub* per facilitare la collaborazione e la revisione del codice. *Git* permette di utilizzare *branch* separati per lo sviluppo parallelo di diverse feature e per la correzione di *bug*. Ogni modifica può essere proposta tramite una *pull request*, che innesca automaticamente una pipeline di test e integrazione. Una volta approvata, la *pull request* può essere integrata nel *branch* principale, attivando ulteriori pipeline di costruzione e rilascio. Questa

struttura permette di mantenere un controllo rigoroso sulle modifiche del codice, garantendo che ogni aggiornamento sia adeguatamente testato e revisionato prima di essere integrato e successivamente distribuito.

7.1.2 Docker

Docker è una piattaforma *open source* progettata per eseguire processi *software* in ambienti isolabili, minimali e facilmente distribuibili, chiamati *container*. Un *container* è un'unità standardizzata e costituisce l'insieme di risorse e dati necessari per eseguire un'applicazione, tra cui librerie, strumenti di sistema, codice e *runtime*. L'entità base tramite la quale vengono definite le caratteristiche di un *container* e i passaggi di costruzione ed esecuzione è detta immagine.

L'utilizzo di *Docker* garantisce che il *software* si comporti in modo uniforme e prevedibile su diversi sistemi operativi e ambienti di sviluppo, test e produzione, assicurando consistenza attraverso tutte le fasi del ciclo di vita del *software*. Questo permette di ridurre i problemi di compatibilità e di semplificare il processo di *deployment*, rendendolo più rapido ed efficiente.

Docker Compose

Docker Compose è uno strumento che permette di definire ed eseguire applicazioni composte da più container *Docker*. Attraverso la definizione di un file in formato Yet Another Markup Language (YAML), è possibile configurare e assemblare i vari processi che compongono l'applicazione, precisando l'ordine di avvio e l'interazione tra di essi.

Docker Compose ha facilitato la definizione delle varie componenti dell'applicazione, permettendo di avviarle e gestirle come un'unica unità.

Docker Hub

Docker Hub è il principale registro pubblico di raccolta e gestione di immagini *Docker*. Connettendosi a questo servizio, è possibile ricercare, pubblicare, condividere e scaricare immagini *Docker*. Contiene un sistema di versionamento e permette di creare *repository*, cioè spazi di condivisione di immagini, pubblici o privati, che possono essere configurati per integrarsi con le *pipeline CI/CD*.

Docker Hub è il registro di immagini *Docker* più popolare e fornisce un supporto integrato con numerosi strumenti di *CI/CD*, tra cui il sottocitato *AWS ECS*.

7.1.3 AWS ECS e EC2

AWS offre diversi servizi per il deployment e la gestione delle applicazioni containerizzate. Tra questi, *ECS* e *EC2* sono fondamentali nel contesto delle *pipeline CI/CD*.

ECS è un servizio di orchestrazione di *container* che facilita la gestione e il dimensionamento di applicazioni in *container*. *ECS* facilita la gestione del ciclo di vita dei *container*, inclusi il deployment, lo scaling e la gestione dello stato delle applicazioni.

ECS può funzionare con due tipi di orchestrazione: *Fargate*, che permette di eseguire i *container* in modalità *serverless*, o con *EC2*, che offre un maggiore controllo sulle risorse di calcolo sottostanti.

Un cluster *ECS* è una logica che raggruppa le risorse di calcolo dove i *container* vengono eseguiti. Un *cluster* può essere composto da istanze *EC2* o da risorse *Fargate*, o una combinazione di entrambe. I *cluster* permettono di organizzare e gestire le risorse di calcolo

necessarie per eseguire i *container*, offrendo visibilità e controllo sulle modalità di esecuzione dei *container*.

In *ECS*, un *task* rappresenta un'istanza di un'applicazione containerizzata in esecuzione. I *task* sono definiti tramite *task definition*, che specificano i dettagli di configurazione del *container*, come l'immagine *Docker* da utilizzare, le risorse richieste, tra cui Control Processing Unit (CPU)$_G$ e memoria, le variabili d'ambiente, i volumi di *storage* e altre configurazioni. Le *task definition* fungono da modello per la creazione e l'esecuzione dei *task*, garantendo che ogni *task* eseguito sia conforme alle specifiche definite. In modo simile al funzionamento di *Docker Compose*, la definizione di un *task* può comprendere l'esecuzione di più *container*.

Quando viene avviato un *task*, *ECS* si occupa di schedularlo su un'istanza del *cluster* disponibile. Se il *cluster* utilizza *EC2*, *ECS* seleziona un'istanza *EC2* con le risorse necessarie per eseguire il *task*. Se il *cluster* utilizza *Fargate*, *AWS* gestisce l'infrastruttura sottostante senza necessità di *provisioning* delle risorse da parte del produttore *software*. *ECS* monitora continuamente lo stato dei *task* e garantisce che il numero desiderato di *task* sia sempre in esecuzione, eventualmente rischedulando i *task* falliti.

EC2 è una piattaforma di elaborazione che fornisce capacità di calcolo scalabile nel *cloud*. Con *EC2*, è possibile avviare macchine virtuali, dette istanze, per eseguire le applicazioni. Le istanze *EC2* possono essere configurate con diverse dimensioni e tipi, permettendo flessibilità nel soddisfare requisiti specifici di elaborazione e memoria.

Datasoil esegue i *container* dei suoi prodotti software in istanze *EC2* poiché il carico e il flusso di dati che queste macchine devono sostenere sono prevedibili e sufficientemente stabili. Utilizzando istanze *EC2* con *ECS*, l'azienda riesce ad ottimizzare l'utilizzo delle risorse di calcolo, gestendo direttamente la configurazione delle istanze.

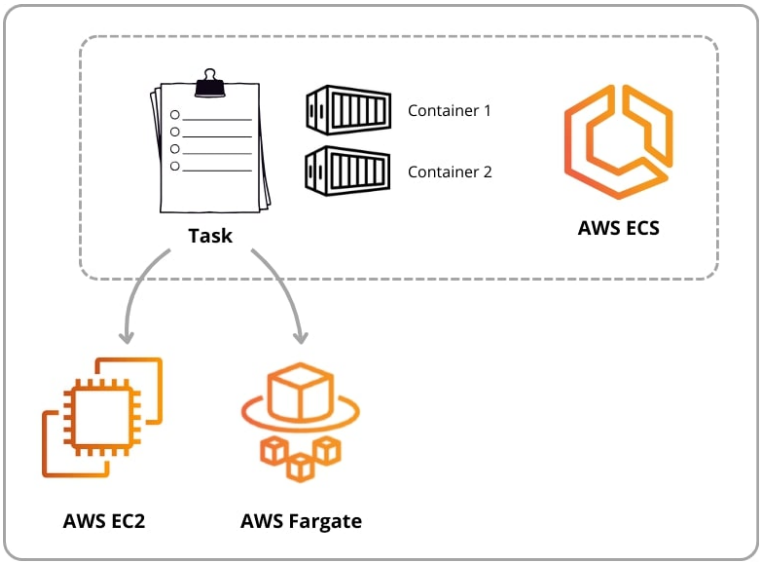

Figura 7.1: Infrastruttura *ECS* con *EC2* e *Fargate*

7.1.4 CircleCI

CircleCI è una piattaforma di *CI/CD* che automatizza i processi di *build*, test e *deployment* del *software*, consentendo di eseguire *pipeline CI/CD* in modo ripetibile ed efficiente.

Utilizzando file di configurazione *YAML*, *CircleCI* permette di definire in dettaglio le *pipeline CI/CD*, specificando ogni passo necessario, chiamato *job*, per costruire, testare e

distribuire l'applicazione. La piattaforma supporta l'esecuzione parallela dei *job*, riducendo significativamente il tempo complessivo richiesto per completare le *pipeline*.

CircleCI è in grado di integrarsi con una vasta gamma di strumenti e servizi, tra cui *Docker*, *AWS* e *GitHub*, semplificando e raggruppando la gestione dell'intero ciclo di vita del *software* in una singola piattaforma. Durante l'esecuzione delle pipeline, questo strumento permette anche di salvare e gestire gli artefatti generati, come file di *log*, report di test e *build* di applicazioni, rendendoli accessibili per ulteriori analisi o per velocizzare esecuzioni successive, ad esempio salvando in una *cache* apposita le dipendenze esterne delle varie componenti.

L'adozione di *CircleCI* in un ambiente *CI/CD* garantisce un'automazione completa del ciclo di sviluppo del *software*, migliorando l'efficienza e la qualità dei processi.

7.2 Ambienti di sviluppo, testing e produzione

L'utilizzo di ambienti distinti per sviluppo, test e produzione permette di isolare le varie fasi del processo di sviluppo, garantendo una separazione chiara tra l'implementazione delle funzionalità, il controllo di qualità e l'uso effettivo da parte degli utenti finali.

Grazie a questa separazione, è possibile proteggere dati sensibili e gestire configurazioni specifiche per ogni ambiente, nonché simulare scenari precisi per valutare la correttezza e le prestazioni del sistema.

Nel contesto del prodotto sviluppato, sono stati definiti i seguenti ambienti:

- Ambiente di sviluppo: utilizzato durante la codifica delle funzionalità dell'applicazione. Questo ambiente è il più flessibile e comprende *debugger* e strumenti di analisi, spesso inclusi dell'Integrated Development Environment (IDE)$_G$ usato, che facilitano la produzione di codice e la correzione di *bug*;

- Ambiente di testing: utilizzato per l'esecuzione dei test, simulando le condizioni dell'ambiente di produzione e isolando il sistema dalle componenti esterne in modo da garantire esecuzioni indipendenti e riproducibili;

- Ambiente di produzione: utilizzato dagli utenti finali. Questo ambiente deve essere stabile, sicuro e performante, garantendo la continuità del servizio e la protezione dei dati.

Per ogni ambiente sono stati definiti file di configurazione nelle componenti *backend* e *frontend*. Questi file consentono di creare distinzioni tra i vari ambienti, specificando parametri come il tipo di autenticazione da utilizzare, collegamenti *URL* ad altri servizi e variabili d'ambiente, cioè variabili globali usate per memorizzare valori di configurazione o informazioni sensibili.

7.3 Configurazione CI/CD tramite CircleCI

Di seguito viene riportata la struttura del file di configurazione di *CircleCI*.

```yaml
version: 2.1

orbs:
  docker: circleci/docker@2.6.0
  aws-cli: circleci/aws-cli@4.1.3

jobs:
  test:
    # definizione del job test
  build_swagger:
    # definizione del job build_swagger
  build_front:
    # definizione del job build_front
  build_back:
    # definizione del job build_back
  deploy:
    # definizione del job deploy

workflows:
  version: 2
  test-build-deploy:
    jobs:
      - test:
        # regole di branching e priorità
      - build_swagger:
        # regole di branching e priorità
      - build_front:
        # regole di branching e priorità
      - build_back:
        # regole di branching e priorità
      - deploy:
        # regole di branching e priorità
```

Codice 7.1: Struttura del file di configurazione di CircleCI.

Come si può notare, il file è composto da quattro sezioni:

- Versione: specifica la versione del file di configurazione di *CircleCI*. La versione 2.1 è attualmente la più recente e include supporto per le *orbs* e la definizione di comandi riutilizzabili all'interno dei *job*;

- *Orbs*: le *orbs* sono pacchetti riutilizzabili di configurazione di *CircleCI* che contengono comandi, *job* e flussi di lavoro predefiniti. Vengono forniti direttamente da *CircleCI* o dalla sua *community* ed evitano di dover ridefinire procedure per connettersi o integrarsi con servizi esterni noti. Tra le *orbs* importate sono presenti `circleci/docker`, per poter utilizzare tutte le funzionalità di *Docker* all'interno di un *job*, e `circleci/aws-cli`, che mette a disposizione l'interfaccia a linea di comando di *AWS* per interagire con i servizi *ECS* ed *EC2*;

- *Jobs*: i *job* sono le unità fondamentali di lavoro in *CircleCI*. Ogni *job* definisce una serie di passaggi da eseguire in un ambiente specifico. Nel file di configurazione vengono

definiti i seguenti *job*:

- `test`: *job* che esegue i test del progetto su un ambiente *Docker*;
- `build_swagger`: *job* che costruisce la documentazione *Swagger* delle *API*;
- `build_front`: *job* che costruisce il *frontend* dell'applicazione su un ambiente *Docker*;
- `build_back`: *job* che costruisce il *backend* dell'applicazione su un ambiente *Docker*;
- `deploy`: *job* che esegue uno *script shell* contenente una serie di comandi *AWS* per distribuire l'applicazione su *ECS*.

- *Workflows*: i *workflow* definiscono l'ordine e la logica di esecuzione dei *job*. Nel file di configurazione viene definito un unico *workflow* chiamato `test-build-deploy` che comprende tutti i *job* definiti in precedenza e si occupa di eseguire la *pipeline* di rilascio dell'applicazione.

7.3.1 Jobs

Di seguito viene riportata la definizione del *job* `build_back`, che ha il compito di costruire la componente di *backend* e rendere disponibile l'artefatto finale agli stadi successivi della *pipeline*.

```
1   build_back:
2     docker:
3       - image: golang:1.22-alpine
4     steps:
5       - checkout
6       - attach_workspace:
7           at: .
8       - restore_cache:
9           keys:
10              - go-mod-v4-{{ checksum "go.sum" }}
11      - run: mkdir dist
12      - run: apk add git openssh
13      - run: git config --global --add
14             url.git@github.com:.insteadOf https://github.com/
15      - run: go build -ldflags="-X main.releaseSHA=$(git rev-parse HEAD)"
16      - save_cache:
17          key: go-mod-v4-{{ checksum "go.sum" }}
18          paths:
19            - '/go/pkg/mod'
20      - run: cp syn-master ./dist/
21      - run: cp config* ./dist/
22      - run: tar -czf syn-master.tar.gz ./dist
23      - persist_to_workspace:
24          root: .
25          paths:
26            - syn-master.tar.gz
```

Codice 7.2: Esempio di definizione di un job.

Oltre a specificare l'immagine *Docker* da utilizzare per il *job*, in questo caso un'immagine leggera che contiene tutto il necessario per compilare un'applicazione *Go*, viene definita la lista di *step* che il *job* deve eseguire:

- `checkout`: *step* predefinito che clona il *repository* corrente all'interno del *job*;
- `attach_workspace`: allega il *workspace* condiviso al percorso specificato, cioè la *directory* corrente;
- `restore_cache`: ripristina un'eventuale salvataggio in *cache* fornito da un'esecuzione precedente utilizzando una chiave basata sul *checksum* del file `go.sum`. Questo *step* permette di evitare di scaricare nuovamente le dipendenze *Go* nel caso in cui il file `go.sum` fosse rimasto invariato;
- `run: mkdir dist`: crea una *directory* `dist` per contenere i file dell'artefatto finale;
- `run: apk add git openssh`: installa i pacchetti `git` e `openssh` necessari per la fase di *build*;
- `run: git config -global -add url.git@github.com: .insteadOf https://github.com/`: configura *Git* per utilizzare il protocollo Secure Shell Protocol (SSH)$_G$ al posto di HyperText Transfer Protocol over Secure Socket Layer (HTTPS)$_G$ per i *repository GitHub*;
- `run: go build -ldflags="-X main.releaseSHA=$(git rev-parse HEAD)"`: compila l'applicazione *Go*, includendo l'*hash* del *commit* corrente come variabile di *build*;
- `save_cache`: salva la *cache* delle dipendenze *Go* utilizzando la stessa chiave del *restore cache*, in modo da velocizzare la fase di installazione delle dipendenze per le *build* future;
- `run: cp syn-master ./dist/`: copia il file binario compilato nella *directory* `dist`;
- `run: cp config* ./dist/`: copia i file di configurazione nella *directory* `dist`;
- `run: tar -czf syn-master.tar.gz ./dist`: crea un archivio `tar.gz` contenente i file della *directory* `dist`, rendendo più veloce e leggera la trasmissione di dati tra un *job* all'altro;
- `persist_to_workspace`: salva l'archivio `tar.gz` creato nel *workspace* condiviso, rendendolo disponibile agli stadi successivi della *pipeline*.

7.3.2 Workflows

L'unico *workflow* contenuto nel file di configurazione si occupa di eseguire in modo ordinato tutti i *job* appena menzionati, oltre a un *job* che sfrutta il comando `publish` dell'*orb* di *Docker* per pubblicare l'immagine dell'intera applicazione sulla piattaforma *Docker Hub*, rendendola disponibile al successivo passaggio di *deploy*, che si collegherà alla piattaforma per recuperare l'immagine, collegandola ad un *task ECS* che verrà eseguito successivamente su una macchina *EC2*.

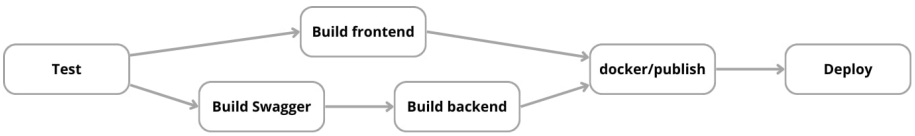

Figura 7.2: Schema di funzionamento del *workflow CircleCI*

Per ciascun *job* riportato nella definizione del *workflow*, sono state impostate delle regole di *branching*, che indicano su quali *branch* il *job* dovrebbe essere eseguito, e delle priorità tra *job*, che definiscono l'ordine in cui questi dovranno essere eseguiti, abilitando eventualmente l'esecuzione di più *job* in parallelo.

Per esempio, il *job* che si occupa di eseguire la pubblicazione dell'immagine *Docker* su *Docker Hub* è abilitato solamente in caso di *push* nel *branch* `main` e `dev`, e deve attendere la terminazione dei *job* `build_front` e `build_back`, che a loro volta dipendono dall'esecuzione dei *job* `test` e `build_swagger`.

Capitolo 8

Conclusioni

8.1 Consuntivo finale

Durante il periodo di stage è stata tenuta una traccia oraria di tutto ciò che è stato svolto, in modo da conoscere precisamente la suddivisione temporale di ogni processo coinvolto.

Di seguito viene riportata una lista che associa ad ogni periodo settimanale l'insieme delle attività svolte.

Periodo	**Attività**
6 maggio - 10 maggio	• Introduzione a *SYN* • Studio del prodotto *SYN* • Studio e analisi degli strumenti e delle tecnologie • Analisi dei requisiti
13 maggio - 17 maggio	• Progettazione *backend* • Progettazione *frontend* • Implementazione struttura *backend* • Implementazione struttura *frontend*
20 maggio - 24 maggio	• Implementazione sistema di autenticazione • Implementazione delle *API* riguardanti i *tenant* • Introduzione a indici di *MongoDB* • Implementazione delle *API* riguardanti gli indici
	Continua nella pagina seguente ...

Tabella 8.1 - Continuo della tabella

Periodo	Attività
27 maggio - 31 maggio	• Studio delle *pipeline* di *build* e rilascio di *SYN* • Introduzione a *CircleCI* • Containerizzazione dell'applicazione per ambienti di sviluppo, test e produzione • Configurazione *CI/CD*
3 giugno - 7 giugno	• Configurazione dell'ambiente di *testing* • Implementazione dei test di unità e integrazione • Documentazione delle *API* tramite *Swagger*
10 giugno - 14 giugno	• Studio *aggregation pipeline* di *MongoDB* • Progettazione delle interfacce dashboard • Implementazione delle *aggregation pipeline*
17 giugno - 21 giugno	• Implementazione delle *dashboard*
24 giugno - 26 giugno	• Revisione del lavoro svolto • Verifica della corretta integrazione con il prodotto *SYN*

Tabella 8.1: Pianificazione delle attività.

In generale, ogni attività affrontata ha avuto lo stesso andamento in termini di produttività, cioè è stata caratterizzata da un periodo iniziale di studio e analisi, che ha rallentato lo sviluppo del prodotto, seguito poi da un periodo di costante miglioramento nell'efficienza produttiva.

Di seguito viene riportato il tracciamento orario delle attività, confrontato a quello stimato nel documento del piano di lavoro.

Durata reale	Durata stimata	Attività
15	15	Formazione aziendale
155	140	Progettazione e sviluppo funzionalità *backend*
75	70	Progettazione e sviluppo funzionalità *frontend*

Continua nella pagina seguente ...

Tabella 8.2 - Continuo della tabella

Durata reale	Durata stimata	Attività
55	75	Progettazione e sviluppo *dashboard* utente

Tabella 8.2: Consuntivo finale.

Nonostante alcuni rallentamenti nello sviluppo della componente *backend*, dovuti all'iniziale inesperienza con le tecnologie utilizzate, un'implementazione efficiente delle *dashboard* ha garantito il rispetto dei tempi prestabiliti.

Analizzando il percorso a valle, una progettazione più attenta e approfondita avrebbe probabilmente permesso di individuare a priori alcuni problemi incontrati, rendendo l'attività di codifica più lineare e deterministica.

8.2 Analisi critica del prodotto ottenuto

8.2.1 Stato di completamento attuale

Il progetto ha raggiunto la soddisfazione di tutti i requisiti iniziali, implementando completamente le specifiche obbligatorie, desiderabili e opzionali.

Il prodotto ottenuto rispecchia le aspettative iniziali, dimostrando di essere un valido strumento per l'azienda nella gestione dei *tenant* associati ai clienti. L'implementazione del sistema ha portato a un significativo miglioramento delle procedure di creazione, modifica e supervisione dei *tenant*, offrendo un contributo essenziale nella gestione della piattaforma *SYN*.

Il prodotto è già completamente integrato all'interno dell'azienda e verrà presto utilizzato per la creazione di nuovi *tenant* durante l'acquisizione di nuovi clienti. L'utilizzo di questo sistema non solo porterà benefici al processo di *onboarding*, ma ridurrà anche in modo significativo la probabilità di errore umano, rendendo le operazioni più sicure e sistematiche.

Un altro vantaggio dell'introduzione del sistema sviluppato è rappresentato dalle nuove possibilità di analisi offerte. Le *dashboard* analitiche integrate consentiranno di monitorare l'andamento dei dati su vari intervalli temporali, fornendo all'azienda una visione dettagliata sull'utilizzo della piattaforma *SYN* da parte dei clienti, permettendo una comprensione più approfondita delle dinamiche operative e dei comportamenti dei clienti, favorendo decisioni strategiche più informate.

Punti di miglioramento ed estensioni future

Nonostante i risultati positivi ottenuti, esistono certamente margini di miglioramento e opportunità per future estensioni. In particolare, le funzionalità di analisi dei dati potrebbero essere ulteriormente approfondite, inserendo una varietà maggiore di grafici e considerando l'integrazione di strumenti di analisi esterni con capacità analitiche superiori, rendendo ancora più efficace la supervisione e l'interpretazione dei dati raccolti.

In prospettiva futura, alcune aree di sviluppo potrebbero includere:

- Confronto tra *tenant*: implementare funzionalità che permettano di confrontare diversi *tenant* per identificare aree di miglioramento e *trend* comuni;

- Calcolo dell'utilizzo: integrare meccanismi per calcolare l'uso delle risorse e proporre tariffe proporzionate all'effettivo consumo;
- Analisi di *trend*: utilizzare tecniche di machine learning$_G$ per identificare e analizzare *trend* specifici, fornendo previsioni e suggerimenti per ottimizzare la piattaforma;
- Sistema di *alert*: sviluppare un sistema di notifiche che avvisi gli amministratori quando si verificano eventi significativi, come il raggiungimento di obiettivi prestabiliti o anomalie operative.

Queste estensioni non solo aumenterebbero la qualità del sistema, ma contribuirebbero anche a consolidare la posizione dell'azienda nel mercato, fornendo ai propri clienti un servizio sempre più preciso, efficiente e personalizzabile.

8.3 Analisi retrospettiva e riflessioni finali

A posteriori, è possibile constatare che le scelte tecnologiche iniziali si sono rivelate corrette, permettendo di implementare la totalità dei requisiti proposti e integrandosi agevolmente nel contesto tecnologico aziendale.

In generale, la complessità delle attività svolte è sempre stata appropriata, risultando sempre in una sfida stimolante, favorevole all'apprendimento incrementale e mai insormontabile o frustrante. Come ogni altra esperienza che include un tasso significativo di novità, la fase iniziale del progetto è stata caratterizzata da una serie di sfide, dovute principalmente all'adozione di nuove tecnologie e all'immersione in un contesto aziendale nuovo. Lo studio del prodotto *SYN* e della sua infrastruttura hanno permesso di entrare maggiormente nell'ottica dell'azienda, consentendo di superare gli ostacoli iniziali. Le sfide affrontate hanno migliorato il mio pensiero critico e affinato le mie abilità di *problem solving*, approfondendo contemporaneamente le mie conoscenze tecniche riguardo i linguaggi di programmazione utilizzati.

Un elemento importante che ha contribuito al successo dell'implementazione è stato il supporto ricevuto dai colleghi, che tramite consigli e delucidazioni hanno svolto un ruolo fondamentale nella risoluzione di problemi tecnici e nel chiarimento di dubbi, rendendo il processo di sviluppo meno tortuoso e più istruttivo. Questo è stato anche reso possibile dal ridotto numero di dipendenti dell'azienda, che ha favorito la mia integrazione, creando un contesto piacevole e stimolante.

Il progetto svolto mi ha permesso di acquisire numerose competenze tecniche e metodologiche, consentendomi di raggiungere risultati tangibili e favorendo al contempo una crescita personale e professionale.

Lo sviluppo di un prodotto *full-stack* ha migliorato la mia comprensione del funzionamento a basso livello delle applicazioni web, consentendomi anche di acquisire familiarità con tecnologie e prodotti popolari nel mondo *enterprise*, spesso di difficile accesso a causa dei loro costi elevati.

Sono consapevole del fatto che, nel mondo del lavoro, è raro dover creare un prodotto da zero, partendo dall'opportunità o dall'idea di base. Inoltre, in un ambiente professionale, ogni attività è spesso svolta da figure diverse, ciascuna con conoscenze verticali rispetto al proprio ambito.

Questa esperienza mi ha permesso invece di vivere ogni attività in prima persona, migliorando la mia comprensione dei processi di ciclo di vita del software e permettendomi di

sviluppare una visione d'insieme che raramente è possibile avere concentrandosi esclusivamente su una parte di un sistema vasto e complesso.

In conclusione, posso affermare di essere soddisfatto del lavoro svolto e grato per l'esperienza vissuta, che ha contribuito alla costruzione di una solida base per lo sviluppo della mia futura carriera professionale.

Bibliografia

Siti

Architettura N-tier. URL: https://it.wikipedia.org/wiki/Architettura_multi-tier.

Architettura three-tier. URL: https://it.wikipedia.org/wiki/Architettura_multi-tier.

Documentazione CircleCI. URL: https://circleci.com/docs/.

Documentazione Cognito. URL: https://docs.aws.amazon.com/cognito/.

Documentazione EC2. URL: https://docs.aws.amazon.com/ec2/.

Documentazione ECS. URL: https://docs.aws.amazon.com/ecs/.

Documentazione Gin. URL: https://gin-gonic.com/docs/.

Documentazione MongoDB Go Driver. URL: https://www.mongodb.com/docs/drivers/go/current/.

Documentazione Prime React. URL: https://primereact.org/.

Documentazione S3. URL: https://docs.aws.amazon.com/s3/.

Integrazione e distribuzione continue. URL: https://www.redhat.com/it/topics/devops/what-is-ci-cd.

Multitenancy. URL: https://learn.microsoft.com/it-it/azure/architecture/guide/multitenant/considerations/tenancy-models.

NoSQL. URL: https://www.oracle.com/it/database/nosql/what-is-nosql/.

Software as a Service. URL: https://www.oracle.com/it/applications/what-is-saas/.

Specifiche REST. URL: https://restfulapi.net/.

Specifiche Swagger. URL: https://swagger.io/docs/specification/about/.