UNIVERSITÀ
DI TRENTO

Descrizione progetto

Progetto di Ingegneria del Software aa.2025/2026

WannaWork

Stefano Videsott
Alessandro Como
Thabo Biagetti

Indice

1. Web APIs

1.1 Specifiche e Architettura

Le API di backend del progetto WannaWork sono state progettate e sviluppate seguendo i rigorosi principi dell'architettura **RESTful**. Esse fungono da livello di comunicazione principale per la Single Page Application (SPA) frontend in Vue.js, garantendo una netta separazione tra la logica di presentazione e la logica di business.

Lo scambio dei payload tra client e server avviene esclusivamente in formato **JSON**. Inoltre, la sicurezza degli endpoint, l'autenticazione e l'autorizzazione basata sui ruoli (Studente e Datore di Lavoro) sono gestite in modalità *stateless* tramite l'utilizzo di **JSON Web Token (JWT)**.

Per la descrizione formale degli endpoint, la definizione dei modelli di dati (struttura delle richieste e delle risposte) e la mappatura dei codici di stato HTTP, è stato adottato lo standard industriale **OpenAPI Specification 3.0** (OAS3). L'adozione di tale standard ha permesso di generare automaticamente una documentazione interattiva, semplificando la fase di testing e l'integrazione continua con l'interfaccia utente.

1.2 Consultazione della Documentazione

Al fine di mantenere questo documento sintetico ed evitare naturali disallineamenti con un codice sorgente in continua evoluzione, la specifica tecnica e dettagliata delle API non viene riportata in forma tabellare in questa sede.

La documentazione completa, interattiva e costantemente allineata all'ultima versione del software, è disponibile e consultabile attraverso i seguenti canali:

- **Swagger UI (Deploy):** L'interfaccia ufficiale generata dinamicamente dal backend dell'applicazione deployata. Permette di esplorare le rotte e testare visivamente ogni singolo endpoint: https://wannawork-app.onrender.com/api/v1/docs/
- **Interactive Documentation (Apiary):** Piattaforma che offre una visualizzazione chiara degli endpoint e permette di simulare le chiamate tramite esempi di mock: wannawork.docs.apiary.io
- **Sorgente OAS3:** Il file di specifica YAML grezzo, che costituisce la *Single Source of Truth* per le API, è versionato insieme al codice sorgente nel repository GitHub ufficiale: WannaWork_App/docs/oas3.yaml

2. Implementation

L'applicazione WannaWork è stata sviluppata adottando lo stack tecnologico **MEVN** (MongoDB, Express.js, Vue.js, Node.js). Questa scelta architetturale, basata interamente sul linguaggio JavaScript sia per il client che per il server, ha permesso di uniformare il processo di sviluppo e di sfruttare la flessibilità del formato JSON per lo scambio di dati end-to-end.

La scelta di queste tecnologie è coerente con il materiale didattico fornito durante il corso e risponde ai requisiti di scalabilità, reattività e manutenibilità richiesti dal progetto.

2.1 Repository Organization

Per garantire un netto disaccoppiamento fisico e logico tra la logica di business e l'interfaccia utente, e per facilitare i flussi di deployment indipendenti, il codice sorgente del progetto è stato diviso e ospitato su **due repository GitHub pubblici e distinti**:

- **Backend API:** github.com/StefanoVidesott/WannaWork_App
- **Frontend SPA:** github.com/StefanoVidesott/WannaWork_Frontend

Nei paragrafi seguenti viene analizzata nel dettaglio la struttura interna di ciascun repository.

2.1.1 Struttura del Repository Backend Il repository dedicato al backend segue i pattern standard per le applicazioni RESTful in Express.js, separando chiaramente la logica di routing, i modelli di dati e i middleware di sicurezza.

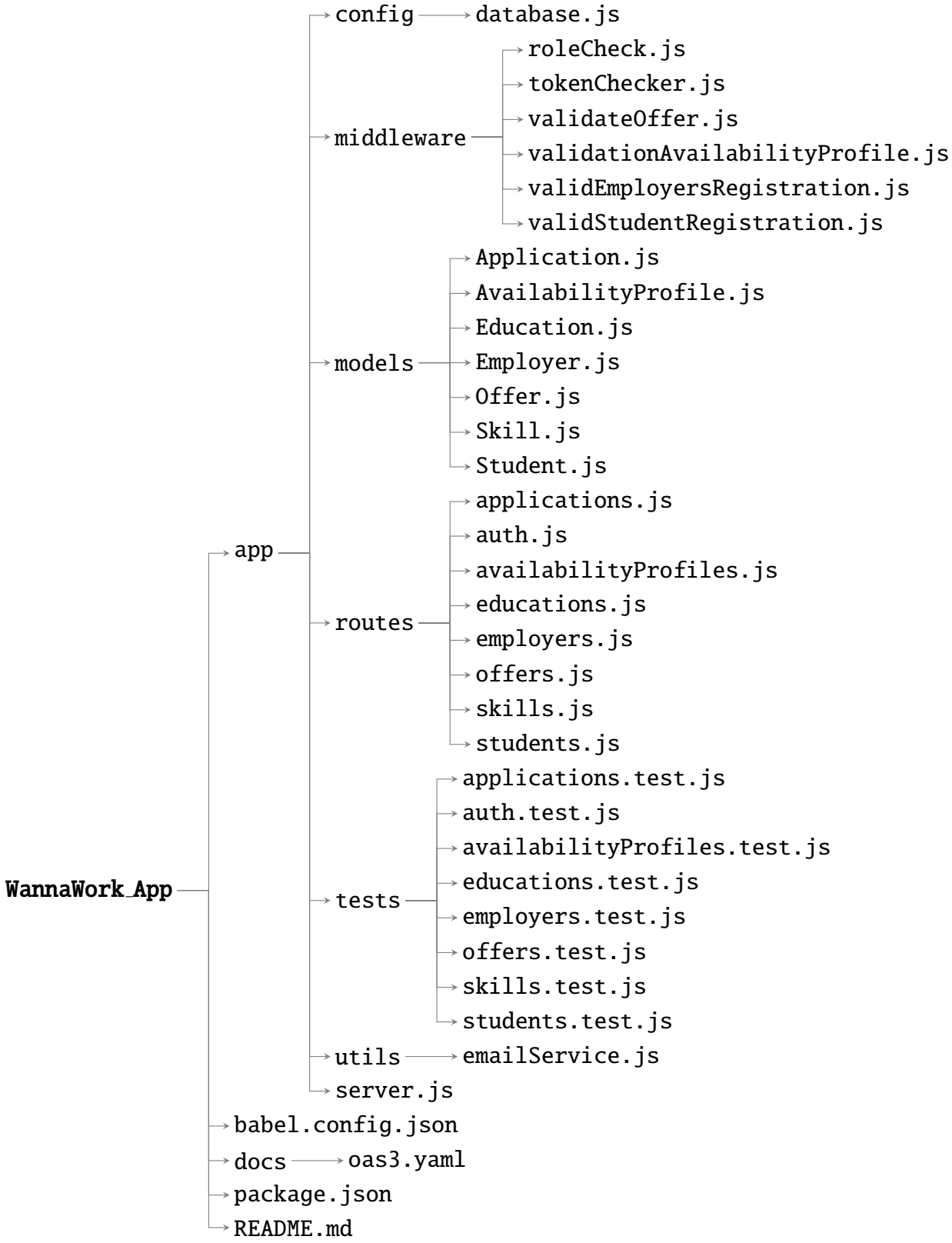

Figura 2.1: *Rappresentazione ad alto livello dell'alberatura del repository Backend.*

File/Directory	Descrizione
app/server.js	Entry point dell'applicazione. Inizializza il server Express, configura i parametri CORS e monta le rotte API.
app/config/database.js	Modulo dedicato alla configurazione e stabilizzazione della connessione con l'istanza MongoDB.
app/models/	Contiene gli schemi Mongoose che definiscono lo strato di persistenza (Application, AvailabilityProfile, Education, Employer, Offer, Skill, Student).
app/routes/	Endpoint REST dell'applicativo. Ogni file isola i controller di un dominio specifico (es. auth.js, offers.js, applications.js).
app/tests/	Suite di test automatizzati (*.test.js) realizzati con Jest e Supertest per verificare il funzionamento degli endpoint, gestire i mock e testare le transazioni.
app/middleware/	Funzioni intermedie per la sicurezza e la validazione. Include il controllo JWT (tokenChecker.js), verifica ruoli (roleCheck.js) e validazione payload (es. validateOffer.js).
app/utils/emailService.js	Servizio di utilità per la gestione e l'invio asincrono di email transazionali (tramite Nodemailer).
docs/oas3.yaml	Specifica API formale in formato OpenAPI 3.0, utilizzata per generare la documentazione Swagger.

2.1.2 Struttura del Repository Frontend Il repository dedicato al frontend ospita la Single Page Application (SPA) realizzata in Vue 3. L'architettura sfrutta la Composition API e il bundler Vite per garantire prestazioni e un'ottima *Developer Experience*.

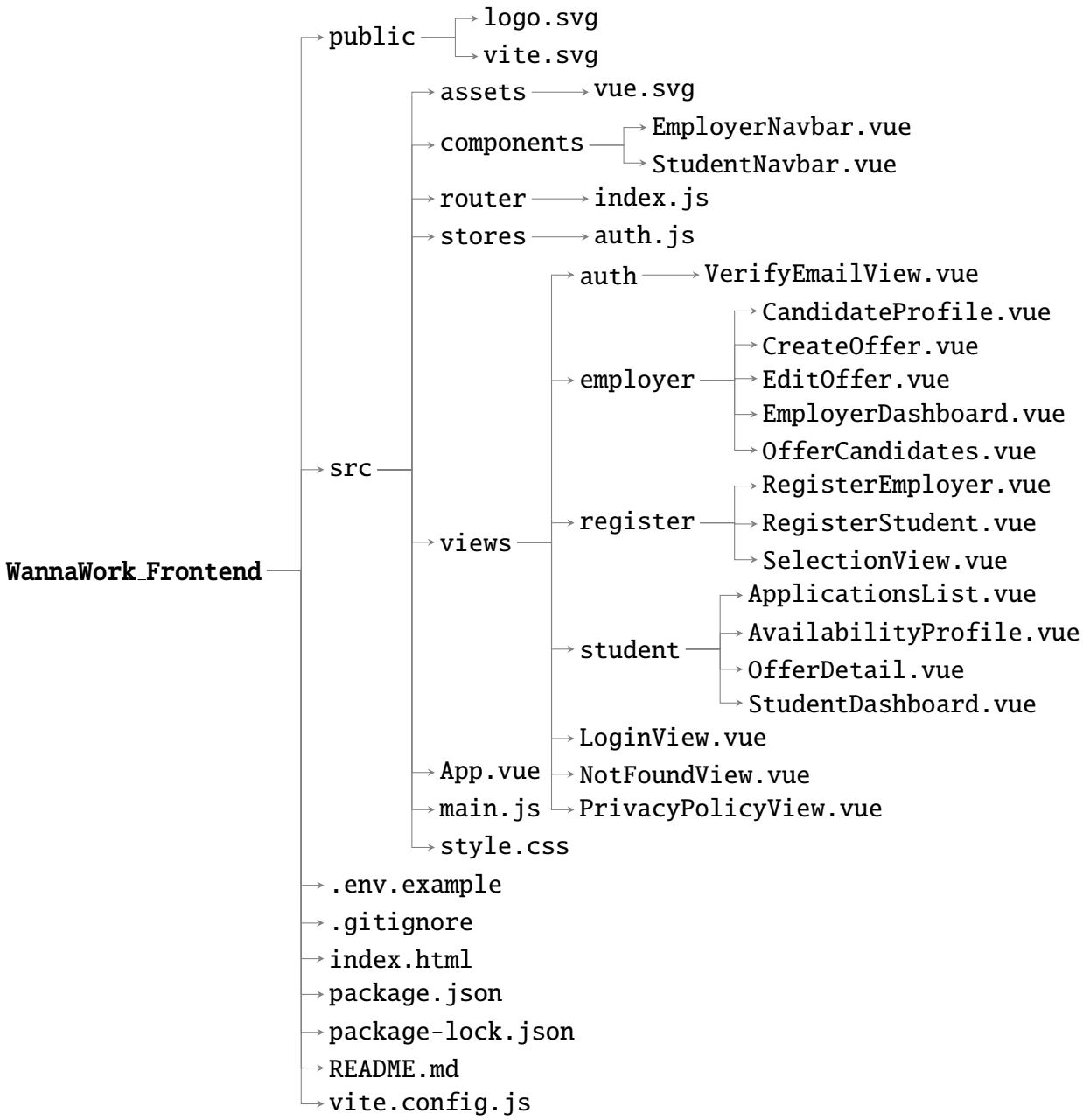

Figura 2.2: *Rappresentazione ad alto livello dell'alberatura del repository Frontend.*

File/Directory	Descrizione
`index.html` & `vite.config.js`	Template HTML principale servito dal browser e configurazione di Vite per la compilazione degli asset.
`src/main.js` & `App.vue`	Entry point dell'applicazione Vue.js. Montano l'istanza principale, il router, lo state manager e includono lo stylesheet globale (`style.css`).
`src/router/index.js`	Configurazione di Vue Router. Definisce la mappa di navigazione e implementa le *Navigation Guards* per proteggere le rotte sensibili.
`src/stores/auth.js`	Store globale gestito tramite Pinia. Centralizza lo stato di autenticazione, il salvataggio dei dati utente e la gestione del JWT.
`src/components/`	Componenti UI riutilizzabili e indipendenti, come le barre di navigazione specifiche per ruolo (`EmployerNavbar.vue`, `StudentNavbar.vue`).
`src/views/`	Core delle interfacce. Diviso funzionalmente in sottocartelle: `auth/` (Verifica Email), `employer/` (Dashboard, Gestione Offerte, `CandidateProfile.vue`), `student/` (Dashboard, Profilo Disponibilità, Candidature) e `register/`. Contiene anche pagine globali come `LoginView.vue`, `PrivacyPolicyView.vue` e `NotFoundView.vue`.

2.2 Branching Strategy e Organizzazione del Lavoro

Lo sviluppo del progetto è stato condotto in modo collaborativo ma modulare, adottando una netta separazione dei compiti basata sulle competenze e sui layer architetturali:

- **Backend:** Como Alessandro e Videsott Stefano si sono occupati della logica lato server, dell'architettura del database MongoDB e della stesura delle suite di testing, suddividendo il lavoro per *feature* funzionali.
- **Frontend:** Videsott Stefano ha curato l'intera implementazione dell'interfaccia utente in Vue.js, la gestione dello stato globale e l'integrazione reattiva con le API REST. Per ottimizzare le tempistiche e permettere agli altri membri del team di concentrarsi su altre fasi del

ciclo di vita del software, lo sviluppo frontend è stato condotto individualmente in locale e poi rilasciato direttamente sul branch `main` a sviluppo concluso.

Per la gestione del versionamento del codice nel **Repository Backend**, è stata adottata una strategia derivata dal **GitFlow Workflow**. Il ramo principale di sviluppo è stato `dev`, dal quale sono stati staccati branch specifici per l'implementazione delle diverse funzionalità, successivamente reintegrati tramite Pull Requests. I branch principali utilizzati sono stati:

- `main`: Codice di produzione stabile, allineato ai deployment.
- `dev`: Ramo di integrazione principale per testare le nuove versioni prima del rilascio.
- `AvailabilityProfile`, `model`, `offers`, `routes`, `test`: Branch tematici utilizzati per isolare lo sviluppo dell'architettura dei dati, l'implementazione degli endpoint e la stesura dei test automatizzati.

2.3 Dependencies

La gestione delle librerie di terze parti per entrambi i repository è affidata al package manager `npm`. Di seguito vengono analizzate le dipendenze critiche definite nei rispettivi file `package.json`, raggruppate per ecosistema:

2.3.1 Dipendenze Backend

Core, Sicurezza e Autenticazione

`express` Framework web minimalista utilizzato per gestire il routing HTTP, i middleware e la logica server-side.

`cors` Middleware per la corretta gestione delle policy *Cross-Origin Resource Sharing*, essenziale per permettere al frontend di comunicare con le API.

`bcryptjs` Libreria utilizzata per l'hashing sicuro delle password prima del salvataggio nel database.

`jsonwebtoken` Implementazione dello standard JWT (RFC 7519) per la generazione e verifica di token firmati, utilizzati per l'autenticazione stateless.

Database e Utilità

mongoose Object Data Modeler (ODM) per MongoDB. Semplifica le interazioni con il database fornendo validazione degli schemi e astrazione delle query.

nodemailer Modulo per l'invio asincrono di email transazionali (es. verifica account, notifiche).

swagger-ui-express e js-yaml Utilizzate per servire la documentazione delle API effettuando il parsing del file `oas3.yaml`.

Dipendenze di Sviluppo e Testing Tra le `devDependencies` backend spiccano **nodemon** (per il riavvio automatico del server durante lo sviluppo) e l'ecosistema di testing formato da **jest** e **supertest**, configurati tramite **babel** per supportare la sintassi ES Modules.

2.3.2 Dipendenze Frontend

Core e State Management

vue (v3) Framework JavaScript progressivo utilizzato per la costruzione di interfacce utente reattive e a componenti.

vue-router Gestore ufficiale del routing per Vue.js, fondamentale per l'implementazione della Single Page Application e per la protezione delle rotte tramite *navigation guards*.

pinia Libreria ufficiale per lo *State Management* di Vue, utilizzata per centralizzare lo stato dell'autenticazione e il token JWT in tutta l'applicazione.

Networking e UI

axios Client HTTP basato su *Promises*, utilizzato per interrogare le API REST del backend in modo asincrono.

tailwindcss e daisyui Framework CSS utility-first e relativa libreria di componenti. Hanno permesso di realizzare rapidamente un'interfaccia utente moderna, coerente e completamente *responsive* senza scrivere CSS personalizzato.

date-fns Libreria leggera per la formattazione e la manipolazione efficiente delle date (es. data di candidatura, validità delle offerte).

2.4 Database

Per la persistenza dei dati è stato scelto **MongoDB**, un database NoSQL orientato ai documenti. L'interazione con il database avviene tramite `mongoose`, che ha permesso di definire schemi rigorosi e relazioni tramite *ObjectId* nonostante la natura *schemaless* di MongoDB, garantendo al contempo flessibilità in fase di prototipazione.

L'architettura dei dati prevede sette collezioni principali fortemente interconnesse per gestire le complesse relazioni tra utenti (Studenti e Datori di lavoro), Offerte pubblicate, Profili di disponibilità e il ciclo di vita delle Candidature. Di seguito viene presentata una panoramica delle singole entità modellizzate:

- **Student**: Gestisce i dati anagrafici, i contatti e le credenziali di accesso degli studenti. Include il tracciamento del consenso alla privacy e si relaziona direttamente con l'entità `Education` per definire il livello di istruzione.
- **Employer**: Rappresenta i datori di lavoro o le aziende iscritte alla piattaforma. Memorizza informazioni istituzionali come la sede principale, il sito web e le credenziali, garantendo un accesso separato rispetto agli studenti.
- **Education**: Un'anagrafica di supporto che censisce in modo univoco gli istituti e le facoltà universitarie. Questo garantisce l'integrità dei dati, standardizzando i percorsi di studio selezionabili dagli utenti.
- **Skill**: Un catalogo centralizzato delle competenze tecniche e trasversali. Funge da punto di congiunzione fondamentale per le logiche di *matching*, venendo referenziato sia dai profili degli studenti sia dai requisiti delle offerte di lavoro.
- **AvailabilityProfile**: Costituisce il curriculum e il profilo di disponibilità dello studente. Raccoglie le competenze possedute (`Skill`), le esperienze pregresse, il monte ore desiderato e le finestre temporali in cui lo studente è disponibile a lavorare.
- **Offer**: Rappresenta gli annunci di lavoro creati dagli `Employer`. Dettaglia la posizione offerta, i requisiti (`Skill`), le condizioni contrattuali e traccia lo stato di pubblicazione dell'annuncio (bozza, pubblicato, scaduto).
- **Application**: Entità transazionale che collega uno `Student`, un'`Offer` e il relativo `Employer`. Gestisce l'intero iter di candidatura, tracciando lo stato corrente (in attesa, revisionata, accettata, rifiutata, ritirata) e mantenendo uno storico temporale di tutti i passaggi di stato.

Le principali strutture dati e le loro interazioni sono riportate nel seguente schema architetturale.

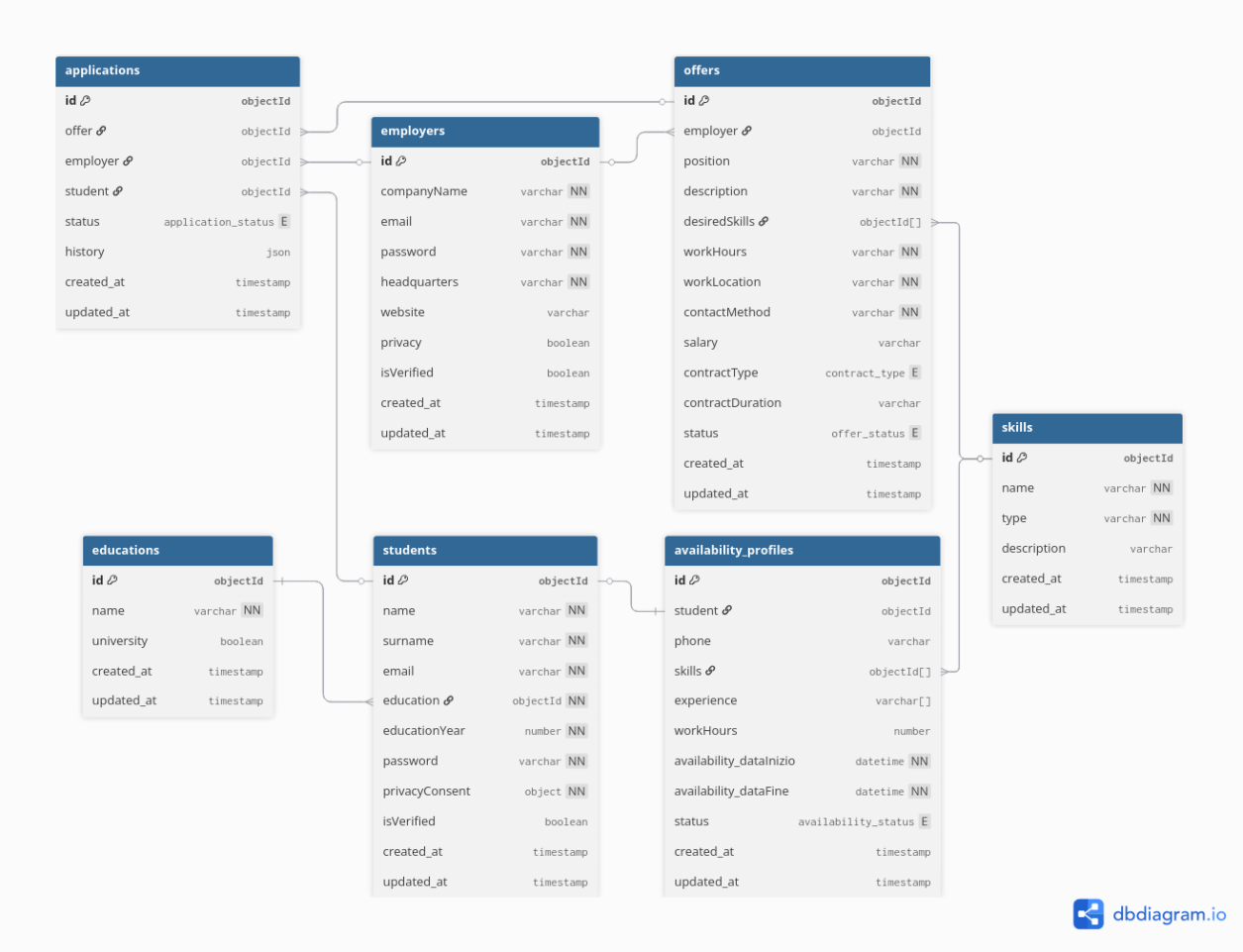

Figura 2.3: *Schema Entità-Relazione (ER) dei modelli Mongoose nel database dell'applicazione.*

2.5 Testing

Per garantire l'affidabilità e la robustezza delle API sviluppate, è stata implementata una suite di test automatizzati che copre in modo estensivo le funzionalità critiche del backend. L'attività di testing si è concentrata sulla verifica degli endpoint REST, simulando le interazioni client-server e validando le risposte HTTP, i codici di stato e la struttura dei payload JSON.

Le tecnologie adottate per questa fase sono state:

- **Jest**: Framework di testing utilizzato come *test runner* e per la gestione delle asserzioni.
- **Supertest**: Libreria per effettuare richieste HTTP fittizie verso l'istanza Express senza dover avviare il server su una porta di rete reale.
- **Cross-env**: Utilizzato per impostare le variabili d'ambiente (es. `NODE_ENV=test`) in modo agnostico rispetto al sistema operativo.

L'implementazione dei test è organizzata in file con estensione `.test.js`, collocati in una directory dedicata (`app/tests/`), replicando fedelmente l'alberatura delle rotte. Questa struttura facilita la manutenzione e l'individuazione immediata dei test associati a ciascun modulo.

Metodologia e Mocking Poiché l'applicazione interagisce con un database reale (MongoDB), per isolare i test unitari ed evitare di "sporcare" il database di produzione o di sviluppo, è stata adottata in modo sistematico la tecnica del **Mocking**. Tramite le funzionalità native di `jest.mock`, i modelli Mongoose (es. `Student`, `Employer`, `Offer`) e i servizi esterni (come l'invio email tramite Nodemailer o la cifratura con `bcryptjs`) sono stati simulati. Questo ha permesso di testare la logica di business e il routing in modo deterministico, estremamente veloce e del tutto indipendente dalla connessione di rete o dallo stato del database.

Attualmente, la suite di test globale comprende **8 test suite** principali per un totale di **62 casi di test** implementati e superati con successo (100% di pass rate).

2.5.1 Casi di Test Di seguito viene riportata una tabella riassuntiva dei casi di test definiti e implementati per ogni modulo e per ogni route.

ID	Descrizione Test Case	Endpoint / Input	Precondizioni (Mock)	Risultato Atteso	Esito
Modulo Autenticazione (`auth.test.js`)					
POST /api/v1/auth/login					
1	Login Studente (Successo)	Payload JSON valido (Email, Password)	Email presente in `Student`. Password valida (`bcrypt`).	Status 200. Restituisce Token JWT e `userType: 'Student'`.	PASS
2	Login Datore (Successo)	Payload JSON valido (Email, Password)	Email assente in `Student`, presente in `Employer`.	Status 200. Restituisce Token JWT e `userType: 'Employer'`.	PASS
3	Utente Non Esistente	Email errata / inesistente	`findOne` restituisce `null` sia per Studente che Datore.	Status 404 Not Found. Messaggio "Email non trovata".	PASS
4	Account Non Verificato	Credenziali corrette	Utente trovato, ma campo `isVerified: false`.	Status 403 Forbidden. Messaggio "Account non verificato".	PASS
5	Password Errata	Email corretta, Password errata	`bcrypt.compare` restituisce `false`.	Status 401 Unauthorized. Messaggio "Password errata".	PASS
6	Credenziali Mancanti	Payload incompleto (es. manca password)	-	Status 400 Bad Request.	PASS
GET /api/v1/auth/verify-email					
7	Verifica Email Studente	Query param: `?token=valid_token`	Token decodificato. Studente trovato con `isVerified: false`.	Status 200. Studente salvato con `isVerified: true`.	PASS
8	Email Già Verificata	Query param: `?token=valid_token`	Token decodificato. Studente trovato con `isVerified: true`.	Status 200. Messaggio "Email già verificata". Nessun salvataggio DB.	PASS

ID	Descrizione Test Case	Endpoint / Input	Precondizioni (Mock)	Risultato Atteso	Esito
9	Verifica Email Datore	Query param: `?token=valid_token`	Studente non trovato. Datore trovato con `isVerified: false`.	Status 200. Datore salvato con `isVerified: true`.	PASS
10	Token Mancante	Nessun query param	-	Status 400 Bad Request. Messaggio "Token mancante".	PASS
11	Token Non Valido / Scaduto	Query param: `?token=bad_token`	`jwt.verify` lancia eccezione.	Status 400 Bad Request. Messaggio "Link non valido".	PASS
12	Utente Non Trovato	Query param: `?token=valid_token`	ID decodificato non corrisponde a nessun record nel DB.	Status 404 Not Found. Messaggio "Utente non trovato".	PASS
Modulo Studenti (`students.test.js`)					
POST /api/v1/students/registration					
13	Registrazione (Successo)	Payload JSON valido (Dati studente e ID Education)	Email non presente nel DB. `Education` esistente.	Status 201 Created. Ritorna redirect. Email di verifica inviata.	PASS
14	Email Già Registrata	Payload JSON valido	`Student.findOne` restituisce un utente esistente.	Status 409 Conflict. Messaggio "Email già registrata".	PASS
15	Istituto (Education) Non Valido	Payload JSON valido	`Education.findById` restituisce `null`.	Status 400 Bad Request. Messaggio "Istituto non valido".	PASS
GET /api/v1/students/can-apply/:offerId					
16	Verifica Candidabilità (Autorizzato)	Query param: `offerId` valido	Profilo visibile, nessuna candidatura attiva, offerta pubblicata.	Status 200 OK. Restituisce `canApply: true`.	PASS
17	Candidabilità Negata (No Profilo)	Query param: `offerId` valido	Profilo disponibilità assente o non visibile.	Status 200 OK. Restituisce `canApply: false` e *reason*.	PASS
18	Candidabilità Negata (Già Candidato)	Query param: `offerId` valido	Candidatura pregressa trovata (`status` diverso da `withdrawn`).	Status 200 OK. Restituisce `canApply: false` e *reason*.	PASS
19	Candidatura Permessa (Dopo Ritiro)	Query param: `offerId` valido	Candidatura pregressa trovata ma con `status: 'withdrawn'`.	Status 200 OK. Restituisce `canApply: true`.	PASS
20	Candidabilità Negata (Offerta Invalida)	Query param: `offerId` valido	Offerta non esistente o con stato diverso da `published`.	Status 200 OK. Restituisce `canApply: false` e *reason*.	PASS
21	ID Offerta Malformato	Parametro URL stringa non-ObjectId	-	Status 400 Bad Request. Messaggio "ID offerta non valido".	PASS
Modulo Profili Disponibilità (`availabilityProfiles.test.js`)					
POST /api/v1/availabilityProfile/create					
22	Creazione Profilo (Successo)	Payload JSON valido (Telefono, Skills, Disponibilità)	`findOne` restituisce `null`. Salvataggio mockato con successo.	Status 201 Created. Messaggio "Profilo creato".	PASS
23	Errore Profilo Esistente	Payload JSON valido	`findOne` restituisce un profilo già esistente per l'utente.	Status 400 Bad Request. Messaggio "Profilo già esistente".	PASS
GET /api/v1/availabilityProfile/me					
24	Lettura Profilo Personale	Header Authorization con JWT studente	`findOne().populate().exec()` restituisce il profilo mockato.	Status 200 OK. Restituisce oggetto `profile`.	PASS

ID	Descrizione Test Case	Endpoint / Input	Precondizioni (Mock)	Risultato Atteso	Esito
25	Profilo Non Trovato	Header Authorization con JWT studente	`findOne` restituisce `null`.	Status 404 Not Found.	PASS
PUT /api/v1/availabilityProfile/:id					
26	Aggiornamento Profilo (Autorizzato)	URL: `/:id`, Body: {`phone`}	Profilo appartiene all'utente (`student == req.user.id`).	Status 200 OK. Modifica confermata e mock `save()` chiamato.	PASS
27	Modifica Profilo Altrui (Vietato)	URL: `/:id`, Body: {`phone`}	Profilo trovato ma associato a uno studente diverso.	Status 403 Forbidden. Messaggio "Non autorizzato".	PASS
DELETE /api/v1/availabilityProfile/:id					
28	Eliminazione Completa	URL: `/:id`, Body: {`password`} corretta	Password valida (`bcrypt`), Transazione MongoDB simulata attiva.	Status 200 OK. Applicazioni ritirate e Datore notificato via email.	PASS
29	Eliminazione Annullata (Pwd Errata)	URL: `/:id`, Body: {`password`} errata	`bcrypt.compare` restituisce `false`.	Status 401 Unauthorized. Transazione NON confermata.	PASS
30	Eliminazione Annullata (Pwd Mancante)	URL: `/:id`, Nessun body	-	Status 400 Bad Request. Messaggio "Password richiesta".	PASS
Modulo Datore di Lavoro (`employers.test.js`)					
POST /api/v1/employers/registration					
31	Registrazione (Successo)	Payload JSON valido (Azienda, Sede, Email)	Email non presente in `Employer`. `bcrypt.hash` mockato.	Status 201 Created. Manda token JWT e redirect.	PASS
32	Errore Email Duplicata	Payload JSON valido	`Employer.findOne` restituisce un documento.	Status 409 Conflict. Messaggio "Email già registrata".	PASS
33	Errore Connessione DB	Payload JSON valido	`save()` lancia un errore generico.	Status 500 Internal Error. Salvataggio fallito.	PASS
34	Errore di Validazione	Payload JSON incompleto o malformato	`save()` lancia eccezione `ValidationError` (Mongoose).	Status 400 Bad Request. Messaggio "Dati non validi".	PASS
35	Tolleranza Fallimento Email	Payload JSON valido	`sendVerificationEmail` lancia un'eccezione.	Status 201 Created. La registrazione ha comunque successo.	PASS
Modulo Educazione / Istituti (`educations.test.js`)					
GET /api/v1/educations					
36	Elenco Istituti (Successo)	Nessun payload. Richiesta `GET`.	`find().select().sort()` restituisce l'array di istituti.	Status 200 OK. Restituisce array JSON con `name` e `university`.	PASS
37	Elenco Vuoto (Successo)	Nessun payload. Richiesta `GET`.	Query Mongoose restituisce un array vuoto.	Status 200 OK. Campo `count: 0` e `data: []`.	PASS
38	Errore Connessione DB	Nessun payload. Richiesta `GET`.	Metodo `find()` lancia eccezione.	Status 500 Internal Error.	PASS
Modulo Competenze (`skills.test.js`)					
GET /api/v1/skills					
39	Elenco Competenze (Successo)	Nessun payload. Richiesta `GET`.	`find().select().sort()` restituisce array di skills.	Status 200 OK. Restituisce JSON con `name` e `type`.	PASS
40	Elenco Vuoto (Successo)	Nessun payload. Richiesta `GET`.	Query Mongoose restituisce un array vuoto.	Status 200 OK. Campo `count: 0` e `data: []`.	PASS
41	Errore Connessione DB	Nessun payload. Richiesta `GET`.	Metodo `find()` lancia eccezione.	Status 500 Internal Error.	PASS
Modulo Offerte (`offers.test.js`)					
POST /api/v1/offers/create					

ID	Descrizione Test Case	Endpoint / Input	Precondizioni (Mock)	Risultato Atteso	Esito
42	Pubblicazione Offerta (Successo)	Payload JSON valido (Posizione, Skills, Salario)	Datore mockato. Metodo `save()` di `Offer` bypassato con successo.	Status 201 Created. Messaggio ”Offerta pubblicata”.	PASS
43	Divieto a Utenti Studenti	Header con JWT avente `userType: 'Student'`	Middleware blocca la richiesta.	Status 403 Forbidden. ”Solo datori di lavoro”.	PASS
44	Profilo Datore Incompleto	Payload valido	Datore trovato ma senza `companyName`.	Status 400 Bad Request. Messaggio ”Completa profilo”.	PASS
GET /api/v1/offers/list					
45	Lista Offerte Paginata	Nessun query param o params: `page`, `limit`, `sort`	Metodi `populate`, `sort`, `skip`, `limit` mockati a cascata.	Status 200 OK. Restituisce array `data` e meta-dati `pagination`.	PASS
GET /api/v1/offers/my-offers					
46	Offerte Personali Datore	Richiesta `GET`	Mock di `find().sort().lean()` e `Application.countDocuments()` unito.	Status 200 OK. Restituisce le offerte con conteggio candidature.	PASS
GET /api/v1/offers/:id					
47	Dettaglio Offerta Esistente	URL param: `/:id` valido	Query mockata con doppio `populate` (Datore e Skills).	Status 200 OK. Oggetto JSON `data` corrispondente all’offerta.	PASS
48	Offerta Inesistente	URL param: `/:id`	`findOne` restituisce `null`.	Status 404 Not Found. ”Offerta non disponibile”.	PASS
PUT /api/v1/offers/:id					
49	Modifica Offerta (Proprietario)	Payload parziale: `{ position: 'New' }`	Offerta trovata appartenente al datore chiamante. Mock di `save()`.	Status 200 OK. Aggiornamento effettuato.	PASS
50	Modifica Offerta (Terzi/Hacker)	Payload parziale	L’`employer` dell’offerta mockata non corrisponde al JWT.	Status 403 Forbidden. ”Non autorizzato a modificare”.	PASS
DELETE /api/v1/offers/:id					
51	Eliminazione con Notifiche	URL: `/:id`, Body: {`reason`}	Transazione MongoDB, aggiornamento `Application` in batch, notifica email.	Status 200 OK. Transazione committata (`commitTransaction`).	PASS
52	Mancanza Motivo Chiusura	Nessun payload nel body	-	Status 400 Bad Request. Transazione terminata.	PASS
53	Rollback per Errore Server	Body: {`reason`}	Metodo `findById` lancia eccezione imprevista (”DB Error”).	Status 500 Internal Error. Esecuzione forzata di `abortTransaction()`.	PASS
Modulo Candidature (`applications.test.js`)					
POST /api/v1/applications/apply					
54	Nuova Candidatura (Successo)	Payload JSON valido (`offerId`)	Profilo studente `visible`, offerta `published`. Mock di `save()` e notifica email.	Status 201 Created. Creata nuova `Application`. Datore notificato via email.	PASS
55	Riattivazione Candidatura	Payload JSON valido (`offerId`)	Trovata candidatura precedente in stato `withdrawn`.	Status 200 OK. Stato riportato a `pending` senza creare duplicati nel DB.	PASS
56	Errore Profilo Non Visibile	Payload JSON valido (`offerId`)	`AvailabilityProfile.findOne` restituisce `null` (o non visibile).	Status 403 Forbidden. Transazione interrotta e candidatura bloccata.	PASS
GET /api/v1/applications/check/:offerId					
57	Verifica Candidatura Attiva	URL param: `/:offerId`	Trovata candidatura per lo studente con stato `pending` o `accepted`.	Status 200 OK. Restituisce `hasApplied: true` e lo status attuale.	PASS

ID	Descrizione Test Case	Endpoint / Input	Precondizioni (Mock)	Risultato Atteso	Esito
58	Verifica Candidatura Ritirata	URL param: `/:offerId`	Trovata candidatura con stato `withdrawn`.	Status 200 OK. Restituisce `hasApplied: false` (ritentabile).	PASS
GET /api/v1/applications/student					
59	Elenco Personale Candidature	Nessun payload. Header JWT Studente.	Mock di `find().populate().sort()` per simulare i dati uniti.	Status 200 OK. Restituisce array `data` con dettagli delle offerte associate.	PASS
PATCH /api/v1/applications/offer/:offerId/withdraw					
60	Ritiro Candidatura (Successo)	URL param: `/:offerId`	Trovata candidatura `pending`. Transazione DB avviata.	Status 200 OK. Stato cambiato in `withdrawn`. Datore notificato via email.	PASS
61	Errore Candidatura Già Valutata	URL param: `/:offerId`	Trovata candidatura con stato `accepted` o `rejected`.	Status 400 Bad Request. Transazione abortita. Messaggio ”Già valutata”.	PASS
62	Ritiro Candidatura Non Trovata	URL param: `/:offerId`	`Application.findOne` restituisce `null`.	Status 404 Not Found. Messaggio ”Nessuna candidatura attiva”.	PASS

3. FrontEnd

Il FrontEnd dell'applicazione WannaWork fornisce le interfacce utente per l'interazione, la visualizzazione e la gestione dei dati. Essendo una piattaforma *multi-tenant*, l'interfaccia è progettata per adattarsi dinamicamente a due tipologie distinte di utenti: gli Studenti e i Datori di Lavoro.

L'applicativo è stato sviluppato come Single Page Application (SPA) utilizzando il framework **Vue.js 3** (Composition API), lo state manager **Pinia** e il bundler **Vite**. Per la prototipazione rapida e la coerenza visiva, l'interfaccia si appoggia al framework CSS utility-first **TailwindCSS** integrato con i componenti di **DaisyUI**.

3.1 Sicurezza e Navigazione (Routing)

La gestione della navigazione è affidata a **Vue Router**. Per garantire la sicurezza delle pagine e impedire accessi non autorizzati, è stato implementato un sistema di *Role-Based Access Control* (RBAC) direttamente nel frontend.

Attraverso l'uso delle *Navigation Guards* (`router.beforeEach`), il sistema verifica costantemente lo stato di autenticazione dell'utente (salvato nello store di Pinia) e i metadati della rotta (`meta: { requiresAuth, role }`). Se un utente non autenticato tenta di accedere a una dashboard, viene reindirizzato al login; allo stesso modo, se un utente tenta di accedere a un'area riservata a un altro ruolo, viene riportato alla propria area di competenza.

3.2 Area Pubblica e Autenticazione

Questa sezione comprende tutte le viste accessibili senza necessità di possedere un token JWT valido. Gestiscono il flusso di ingresso, l'onboarding dei nuovi utenti e la consultazione di documenti legali come la Privacy Policy.

Login (`LoginView`) L'entry point dell'applicazione (`/`). Permette all'utente di inserire le proprie credenziali per accedere al sistema.

Dal punto di vista implementativo, il componente si appoggia allo store di autenticazione (`useAuthStore`) per inviare le credenziali al backend. In caso di successo, il sistema riconosce automaticamente il ruolo dell'utente esaminando il payload della risposta (`userType`) e lo reindirizza dinamicamente alla rispettiva dashboard (`/dashboard/student` o `/dashboard/employer`).

Il design è stato recentemente aggiornato per adottare un layout centrato e moderno, inserendo il logo vettoriale dell'applicazione in cima alla form per rafforzare la *brand identity*.

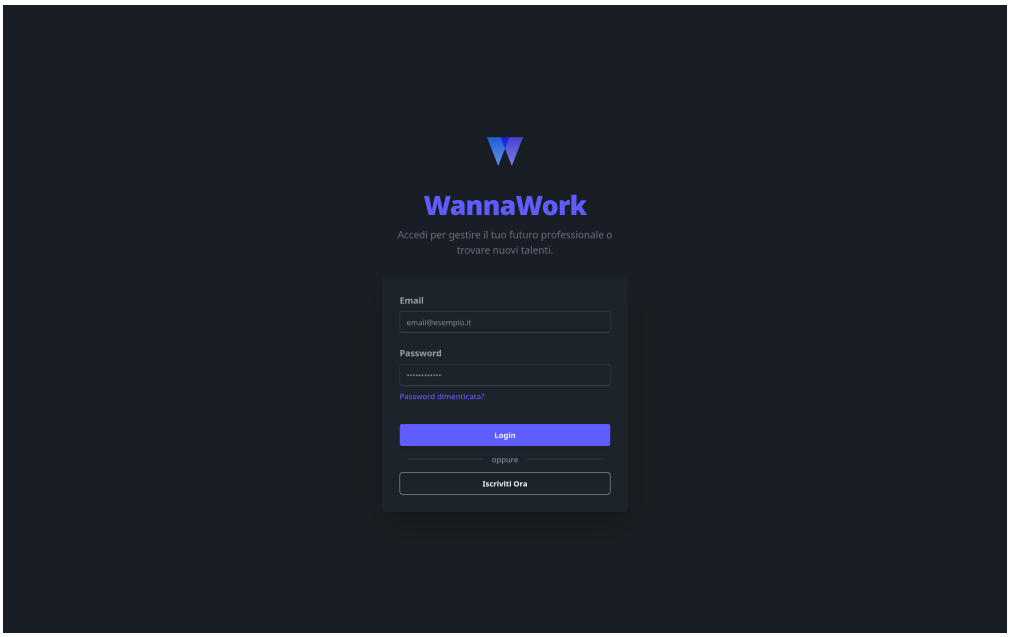

Figura 3.1: *Schermata principale di Login in stato di attesa con il nuovo layout centrato.*

Il componente è stato progettato ponendo particolare attenzione alla *User Experience* (UX):

- **Stato di caricamento:** Durante l'attesa della risposta dal server, il pulsante di login viene disabilitato e mostra uno *spinner* animato, prevenendo invii multipli accidentali (`loading.value`).
- **Gestione degli errori:** Qualora l'autenticazione fallisca (es. password errata, account non verificato o utente inesistente), il blocco `try-catch` intercetta l'eccezione e innesca la comparsa di un banner di errore nativo di DaisyUI (`alert-error`), fornendo un feedback testuale immediato all'utente.

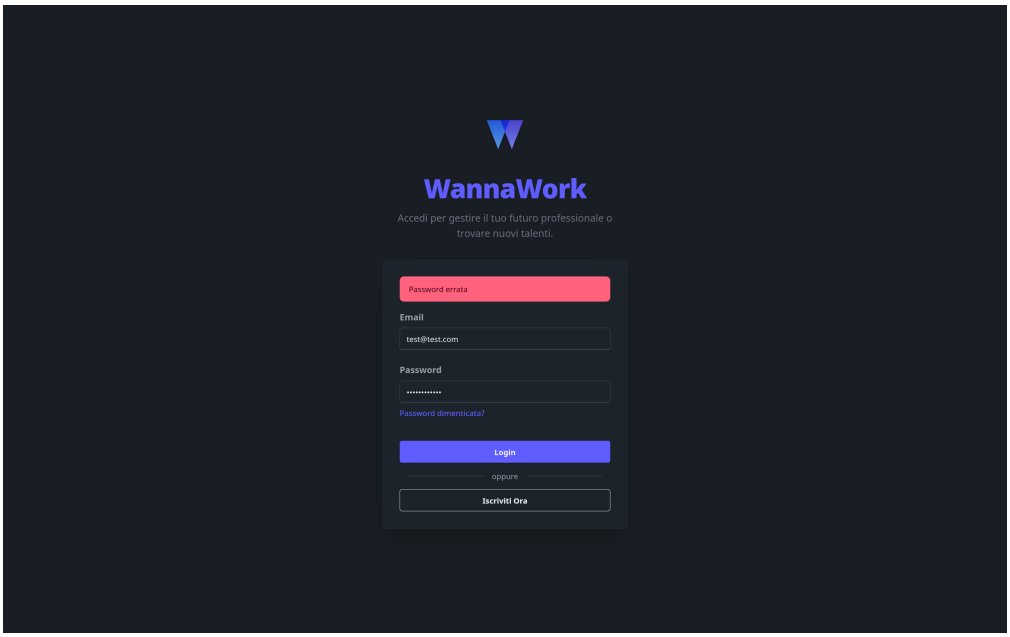

Figura 3.2: *Schermata di Login che mostra la gestione reattiva di un errore di autenticazione.*

Selezione Registrazione (`SelectionView`) Accessibile tramite il pulsante ”Iscriviti Ora” presente nella schermata di login, questa vista gestisce il primo step (onboarding) per gli utenti non ancora registrati.

Non essendo presenti logiche applicative o chiamate di rete in questa fase, lo sforzo progettuale si è concentrato esclusivamente sulla *User Experience* (UX) e sulla chiarezza visiva. La pagina presenta all’utente una scelta dicotomica, immediata e inequivocabile, tra i due percorsi di profilazione supportati dalla piattaforma: ”Studente” o ”Datore di Lavoro”.

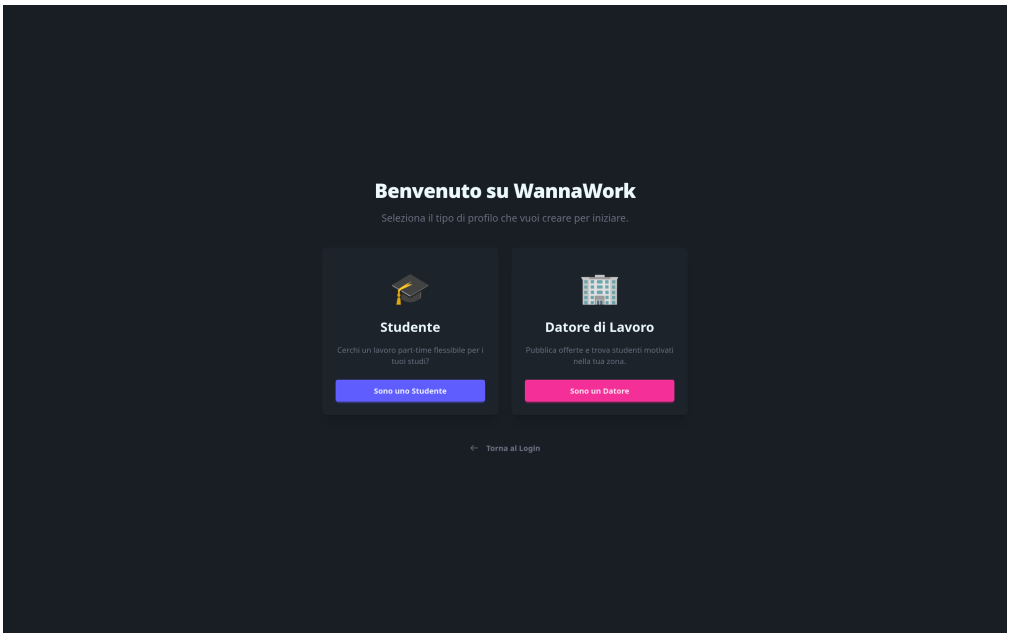

Figura 3.3: *Schermata di selezione della tipologia di account in fase di registrazione.*

Dal punto di vista dell'implementazione:

- **Routing Dichiarativo:** La navigazione verso i moduli di registrazione specifici (`/register/student` o `/register/employer`) è gestita nativamente tramite il componente `<router-link>`, garantendo una transizione istantanea senza ricaricamento della pagina.
- **Design Responsivo:** Grazie alle classi di **TailwindCSS** (`flex-col md:flex-row`), i due pannelli si affiancano orizzontalmente su schermi desktop, mentre si dispongono verticalmente su dispositivi mobili per garantire la leggibilità.
- **Feedback Visivo:** L'interfaccia sfrutta le *Card* di **DaisyUI** con un effetto *hover* personalizzato (bordi colorati `primary` o `secondary`) che evidenzia la selezione dell'utente al passaggio del mouse, migliorando l'interattività percepita.

Registrazione Studente (`RegisterStudent`) Questa vista ospita il modulo per la registrazione di un nuovo profilo Studente. Richiede l'inserimento dei dati anagrafici, dell'email (possibilmente istituzionale), la selezione dell'istituto di appartenenza e la configurazione delle credenziali d'accesso.

Dal punto di vista dell'implementazione front-end, il form presenta una serie di automatismi mirati a migliorare l'esperienza utente e prevenire l'invio di dati errati al server:

- **Caricamento asincrono dati correlati:** Al momento del caricamento (`onMounted`), il componente effettua una chiamata asincrona tramite *Axios* all'endpoint `/api/v1/educations` per popolare dinamicamente il menu a tendina degli istituti scolastici/universitari disponibili. Durante il recupero dei dati, il campo viene disabilitato e mostra un indicatore visivo di caricamento.
- **Validazione reattiva della Password:** Tramite le variabili computate di Vue (`computed`), il sistema valuta in tempo reale la lunghezza della password (minimo 12 caratteri) e la corrispondenza esatta con il campo di conferma, colorando dinamicamente i bordi degli input (verde in caso di successo, rosso in caso di errore) e abilitando il pulsante di sottomissione solo a condizioni soddisfatte.

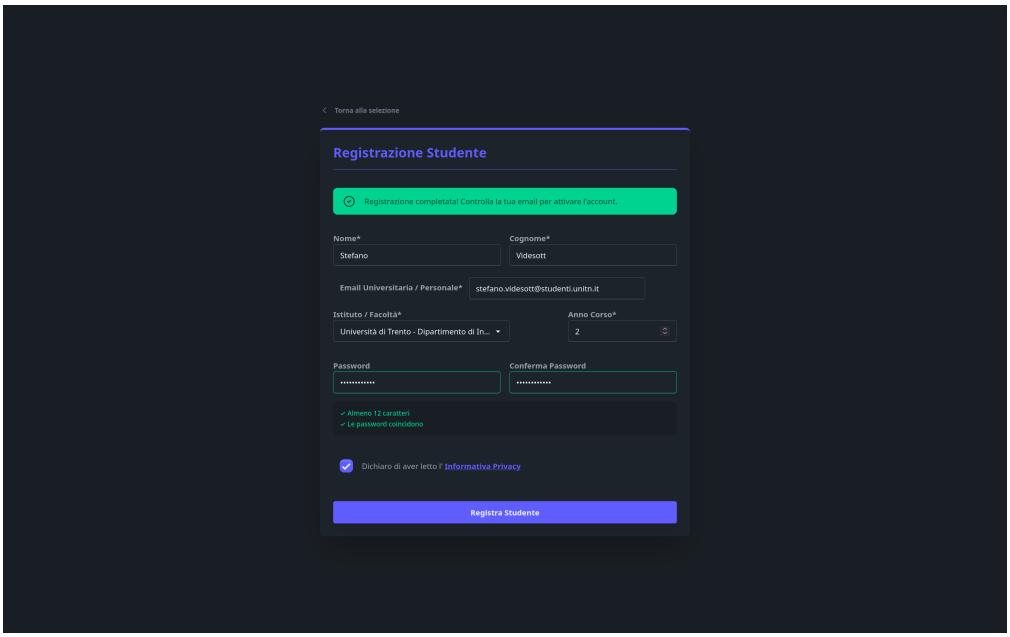

Figura 3.4: *Modulo di registrazione Studente compilato correttamente, con bordo e accenti del colore tematico primario (blu).*

Un controllo ulteriore è applicato all'accettazione dell'informativa sulla privacy. Qualora l'utente tenti di sottomettere il modulo senza aver spuntato l'apposita casella (obbligatoria ai sensi del GDPR), l'invio viene bloccato a livello client e l'interfaccia si adatta dinamicamente, tingendo il box di rosso per evidenziare la dimenticanza.

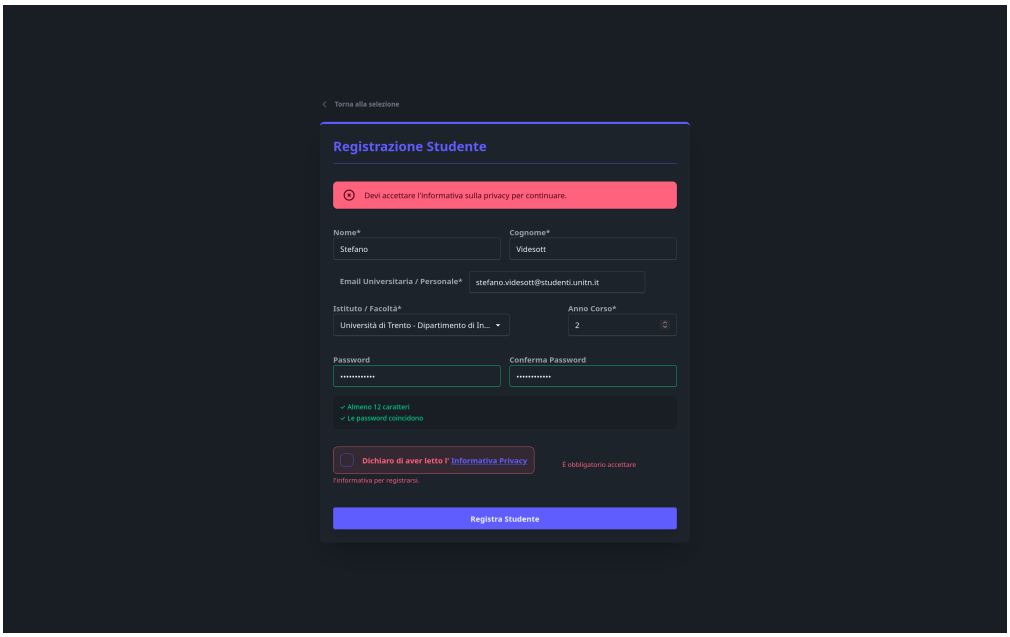

Figura 3.5: *Evidenziazione dinamica dell'errore per mancata accettazione della Privacy Policy.*

Infine, in caso di successo della chiamata di registrazione (`POST /api/v1/students/registration`), il sistema mostra un avviso verde di conferma e avvia un timer che, dopo 5 secondi, reindirizza automaticamente l'utente alla schermata di login.

Registrazione Datore di Lavoro (`RegisterEmployer`) Questa vista gestisce l'onboarding per le aziende e le attività commerciali. Il modulo richiede l'inserimento dei dati essenziali per la creazione del profilo aziendale: Ragione Sociale (Nome Azienda), Email di contatto, indirizzo della Sede Principale e, opzionalmente, il link al Sito Web ufficiale.

L'interfaccia mantiene una stretta coerenza visiva con il modulo studenti ma utilizza i colori semantici differenziati per l'area aziendale (es. `text-secondary`, bordo superiore fucsia) e implementa logiche di validazione reattiva customizzate per il dominio business:

- **Validazione Indirizzo Sede:** Tramite una computed property di Vue (`isHeadquartersValid`), il sistema verifica in tempo reale che l'indirizzo inserito sia lungo almeno 10 caratteri, garantendo l'acquisizione di indirizzi completi (necessari successivamente per la geolocalizzazione o filtri delle offerte). Se il criterio non è soddisfatto, il campo si tinge di rosso e mostra un messaggio di avviso (`label-text-alt text-error`).
- **Validazione URL Sito Web:** Sebbene opzionale, se il campo sito web viene compilato, il sistema utilizza il costruttore nativo JavaScript `new URL()` per assicurarsi che il formato sia valido e che includa i protocolli corretti (`http://` o `https://`).

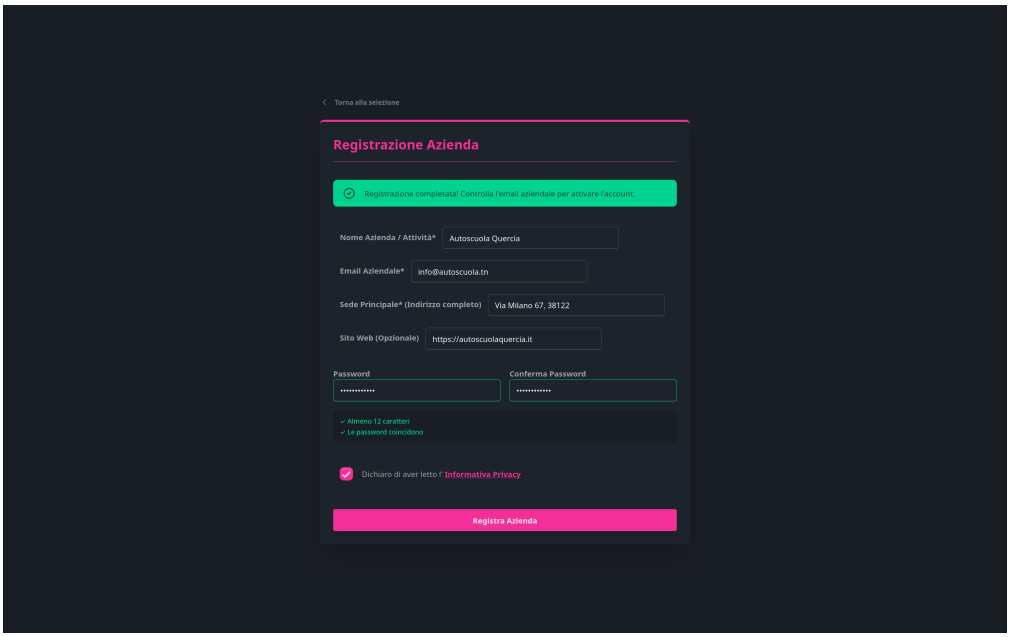

Figura 3.6: *Modulo di registrazione per il Datore di Lavoro con campi validati correttamente.*

Come per lo studente, anche questo form è protetto dalla validazione in tempo reale della complessità della password (minimo 12 caratteri e corrispondenza di conferma) e dal vincolo bloccante sull'accettazione esplicita della Privacy Policy. Se le condizioni non sono soddisfatte, il pulsante di "Registra Azienda" rimane disabilitato, impedendo chiamate di rete fallimentari verso le API (`POST /api/v1/employers/registration`).

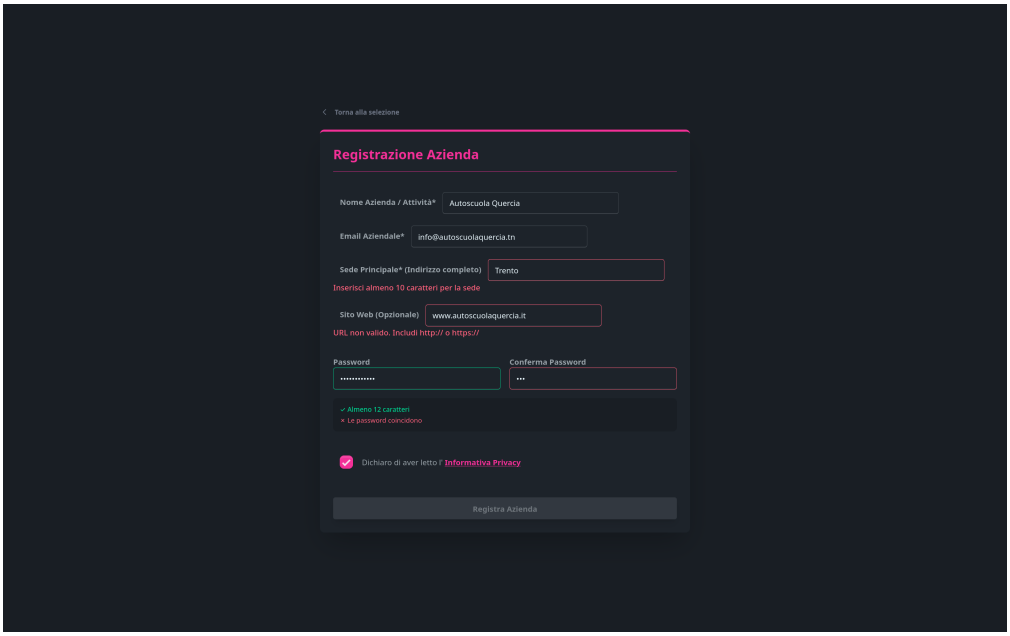

Figura 3.7: *Evidenziazione dinamica degli errori di validazione per i campi Sede, Sito Web e Conferma Password.*

3.2.1 Privacy Policy e Disclaimer (`PrivacyPolicyView`) Accessibile tramite un link dedicato presente in tutti i moduli di registrazione, questa vista ha il duplice scopo di presentare l'Informativa sulla Privacy (simulata ai sensi del GDPR) e, soprattutto, di tutelare il team di sviluppo attraverso un disclaimer accademico.

La pagina si apre evidenziando immediatamente un *Alert* di colore giallo (`alert-warning`), il quale informa esplicitamente l'utente finale (solitamente un docente o un tester) della natura puramente didattica del software, scoraggiando attivamente l'inserimento di dati personali reali o password utilizzate per altri servizi.

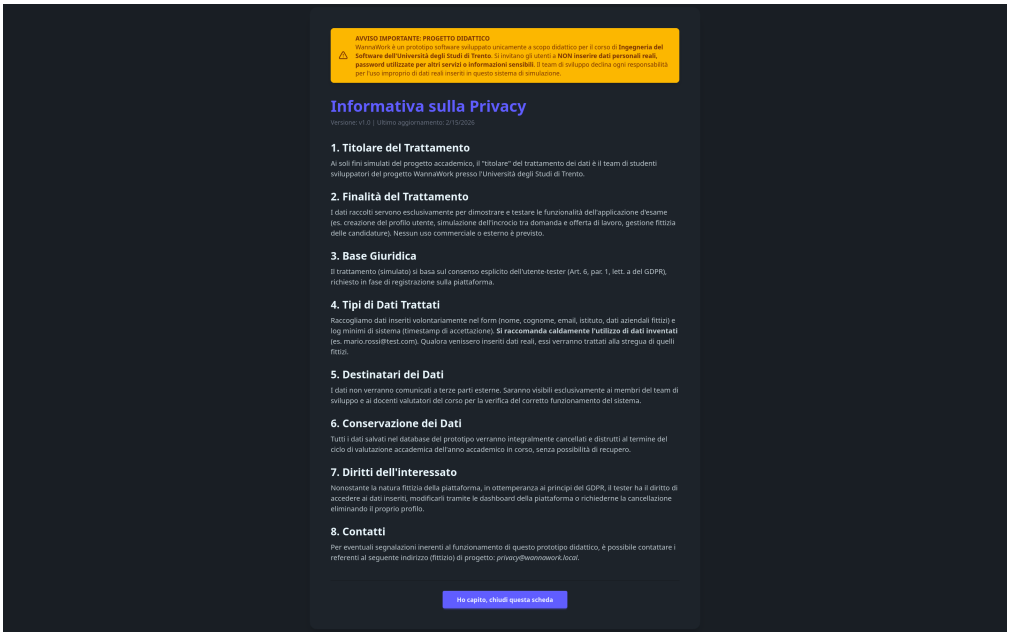

Figura 3.8: *Schermata della Privacy Policy con l'evidente disclaimer per il progetto universitario.*

Il contenuto testuale dell'informativa è strutturato in sezioni chiare (Titolare, Finalità, Conservazione) per simulare in tutto e per tutto un vero documento legale. Dal punto di vista dell'implementazione tecnica in Vue.js:

- **Gestione Navigazione Finestra:** Poiché il documento viene aperto tipicamente in una nuova scheda del browser (`target="_blank"`), il pulsante di chiusura implementa una funzione mista. Invocando `window.close()`, tenta di chiudere la scheda corrente; qualora le policy del browser lo impedissero (ad esempio se la pagina è stata aperta nella stessa scheda o ricaricata), un `setTimeout` esegue un fallback sicuro sfruttando `window.history.back()` per riportare l'utente al modulo di registrazione.

3.2.2 Verifica Email (`VerifyEmailView`) Questa vista funge da pagina di transizione e landing page per l'utente che clicca sul link di conferma ricevuto via email al termine della registrazione

(sia esso Studente o Datore di Lavoro).

Il componente `VerifyEmailView` è progettato attorno a un sistema a stati reattivo (`status: 'loading' | 'success' | 'error'`), che governa la renderizzazione condizionale dell'interfaccia.

Il flusso logico, eseguito nel momento in cui il componente viene montato nel DOM (`onMounted`), si articola nei seguenti passaggi:

1. **Estrazione del Token:** Viene recuperato il parametro `token` dalla query string dell'URL tramite l'hook `useRoute()` di Vue Router. Se il token è assente, l'elaborazione si interrompe immediatamente, passando allo stato di errore.

2. **Validazione Asincrona:** In presenza del token, l'applicazione effettua una chiamata `GET` all'endpoint `/api/v1/auth/verify-email`. Durante questa frazione di secondo, l'interfaccia mostra lo stato `'loading'`, rassicurando l'utente con uno spinner animato generato da DaisyUI.

3. **Feedback e Reindirizzamento:**

   - **Successo:** Se il server convalida l'account (`response.data.success === true`), lo stato passa a `'success'`. L'interfaccia mostra un'icona di spunta verde (*Checkmark*) e un messaggio di benvenuto. In background, un timer `setTimeout` attende 3 secondi prima di forzare il reindirizzamento dell'utente (`router.push('/')`) verso la schermata di Login, permettendogli di leggere il messaggio.
   - **Errore:** Se il token risulta scaduto, manomesso o l'account è già stato verificato, il blocco `catch` intercetta la risposta del backend e imposta lo stato su `'error'`. Viene mostrata un'icona di errore rossa accompagnata dal messaggio di fallimento specifico restituito dall'API. L'utente viene invitato a tornare manualmente alla Home Page tramite un pulsante dedicato.

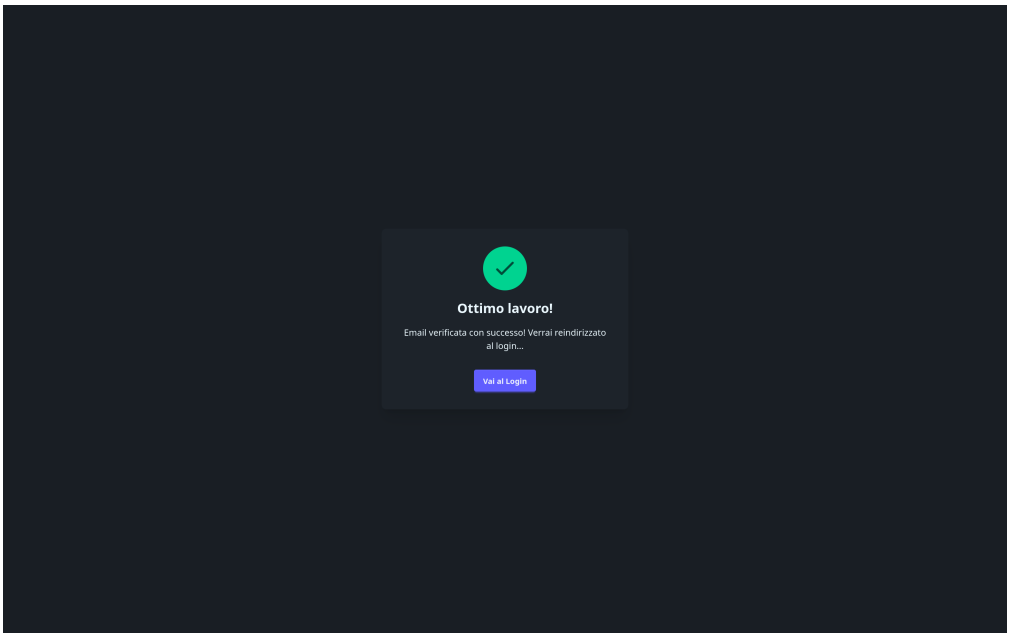

Figura 3.9: *Schermata di successo a seguito della verifica di un account tramite token JWT valido.*

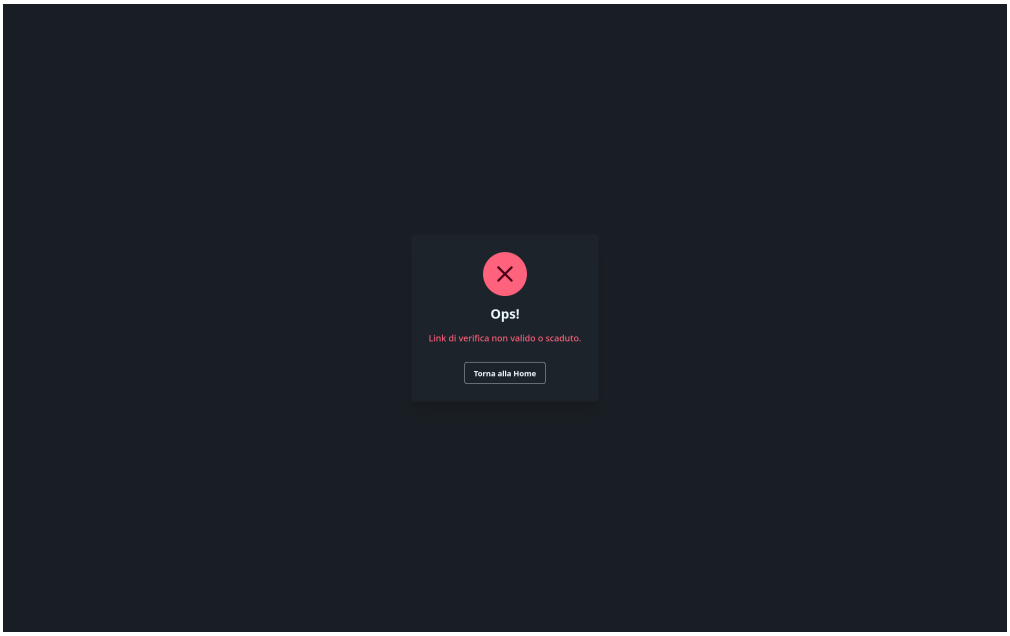

Figura 3.10: *Gestione di un token di verifica scaduto o assente.*

3.2.3 Pagina Non Trovata (`NotFoundView`) Nel contesto di una Single Page Application, è essenziale gestire le navigazioni verso percorsi (URL) inesistenti, digitati erroneamente dall'utente o non più disponibili. Questa vista funge da *catch-all route* (rotta di fallback) configurata nel router di Vue tramite l'espressione regolare `/:pathMatch(.*)*`.

Il design del componente è focalizzato sul recupero dell'errore (Error Recovery) e si presenta con un'estetica amichevole, caratterizzata da un grande "404" animato (`animate-bounce`) e un messaggio esplicativo chiaro.

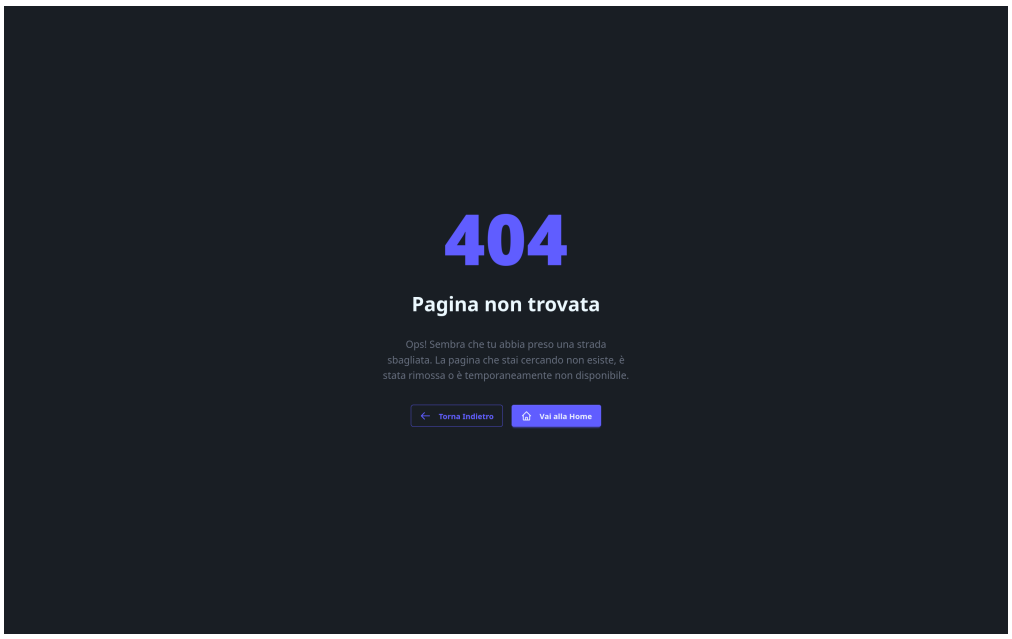

Figura 3.11: *Pagina personalizzata per l'errore 404 (Not Found) con azioni di ripristino navigazione.*

Dal punto di vista logico, l'interfaccia non si limita a un semplice avviso passivo, ma offre all'utente due percorsi di uscita calcolati dinamicamente:

- **Torna Indietro:** Sfruttando la History API nativa del browser tramite `useRouter().back()`, questo pulsante permette all'utente di ritornare all'esatta schermata precedente alla generazione dell'errore, senza perdere l'eventuale stato dell'applicazione.
- **Vai alla Home (Routing Intelligente):** Il pulsante "Home" non reindirizza ciecamente alla pagina di login. Sfruttando una *computed property* (`homeRoute`) che interroga lo store di autenticazione (`useAuthStore`), il componente valuta il ruolo dell'utente: se è loggato come Studente lo riporta alla `/dashboard/student`, se come Datore alla `/dashboard/employer`, e solo se non è autenticato lo riporta alla root `/`. Questo dettaglio garantisce una *User Experience* fluida e senza interruzioni, rispettando le policy di accesso (RBAC).

3.3 Area Studente

L'Area Studente rappresenta il nucleo operativo per gli utenti in cerca di occupazione. Una volta superata la fase di autenticazione, il sistema carica le interfacce riservate a questo ruolo, garantendo

l'accesso alle funzionalità di consultazione, candidatura e gestione del proprio profilo.

3.3.1 Barra di Navigazione (`StudentNavbar`) Il layout di tutte le viste dell'Area Studente è unificato dalla presenza di una barra di navigazione dedicata (`StudentNavbar`), posizionata in testa alla pagina (*sticky header*) per garantire un'accessibilità costante.

Il componente, oltre a mostrare il logo e il badge identificativo del ruolo, fornisce i link diretti alle sezioni principali: *Offerte*, *Le mie Candidature* e *Profilo Disponibilità*. Il sistema di routing nativo di Vue aggiorna dinamicamente lo stile del link attivo (`active-class="active font-bold"`) per fornire un feedback contestuale all'utente.

In ottemperanza ai requisiti di progetto (User Stories), la barra include anche le voci di menu per **Ricerca offerte** e **Offerte preferite**. Poiché tali funzionalità non sono previste per il rilascio corrente, i relativi pulsanti sono stati implementati in stato disabilitato. Sfruttando le classi di DaisyUI e Tailwind (`disabled`, `cursor-not-allowed`, `text-base-content/50`), l'interfaccia comunica visivamente all'utente che l'azione non è permessa, mostrando al passaggio del mouse un avviso esplicito (tramite l'attributo HTML `title`) che indica la natura *Work In Progress* (WIP) del componente.

Sul lato destro, un menu a tendina (*dropdown*) associato all'Avatar dell'utente (generato tramite l'API `ui-avatars.com` a partire dal nome) ospita la medesima struttura di navigazione ottimizzata per dispositivi mobili e il pulsante di Logout, il quale si occupa di invalidare il token nello store di Pinia e reindirizzare l'utente all'Area Pubblica.

Figura 3.12: *Dettaglio della* `StudentNavbar` *con link attivi e link disabilitati per funzionalità future.*

3.3.2 Dashboard Studente (`StudentDashboard`) La Dashboard Studente funge da landing page per l'utente autenticato e ospita il catalogo completo e aggiornato delle offerte lavorative pubblicate dalle aziende sulla piattaforma WannaWork.

L'interfaccia è stata ingegnerizzata per gestire in modo efficiente elenchi di dati potenzialmente lunghi, adottando un approccio moderno alla *Data Fetching* e alla *User Experience*:

- **Recupero e Paginazione Reattiva:** Le offerte non vengono caricate in blocco, ma interrogate tramite l'endpoint `GET /api/v1/offers/list`. Il componente sfrutta la reattività di `vue-router` (grazie al *watcher* su `route.query`) per ricaricare automaticamente i dati ogni qualvolta l'utente interagisce con i controlli di ordinamento ("Più recenti" / "Meno recenti") o naviga tra le pagine tramite il componente *pagination* posto a fondo pagina.

- **Skeleton Loader (Stato di Attesa):** Per mitigare la latenza di rete durante la chiamata API, il sistema non mostra una schermata vuota ma implementa uno *Skeleton Loader* animato (`animate-pulse`). Questo *pattern UI* riproduce la sagoma sfumata delle *Card* delle offerte, comunicando all'utente che l'app è in fase di elaborazione dati senza causare disorientamento visivo (*Cumulative Layout Shift*).
- **Formattazione Temporale:** La data di pubblicazione dell'offerta viene formattata lato client in linguaggio naturale (es. "circa 2 ore fa") tramite la libreria `date-fns`, migliorando sensibilmente la leggibilità rispetto ai classici timestamp.

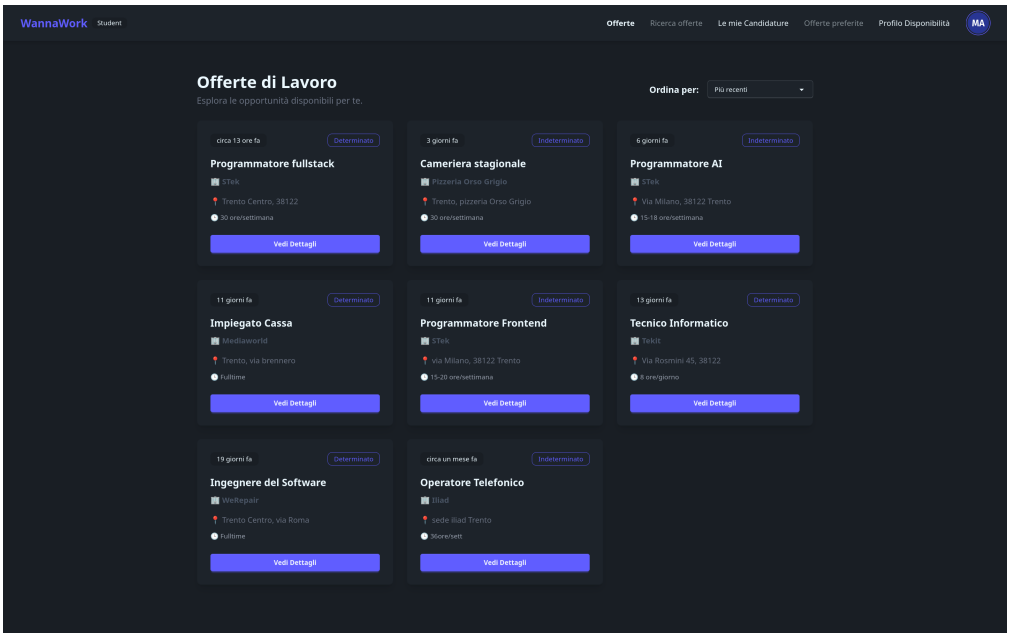

Figura 3.13: *Dashboard Studente con elenco delle offerte in formato Card.*

Cliccando su una singola *Card*, l'utente viene reindirizzato dinamicamente (`goToDetail`) verso la vista di approfondimento dell'offerta specifica, passando l'identificativo univoco dell'annuncio (`_id`) come parametro di rotta.

3.3.3 Dettaglio Offerta e Candidatura (`OfferDetail`)

Questa vista rappresenta il punto focale dell'interazione per lo Studente. Consente di visualizzare in dettaglio tutte le informazioni relative a un annuncio lavorativo e fornisce l'interfaccia per inviare o ritirare una candidatura.

Dal punto di vista architetturale e logico, il componente gestisce uno stato interno piuttosto complesso, orchestrando chiamate multiple verso il backend in fase di montaggio (`onMounted`):

1. **Recupero Dati Offerta:** Effettua una chiamata `GET` all'endpoint `/api/v1/offers/:id` per ottenere la descrizione, i requisiti (*Skills*), il salario e le info aziendali.

2. **Verifica Stato Candidatura (`checkApplicationStatus`):** Contatta parallelamente l'endpoint `/api/v1/applications/check/:id` per determinare se l'utente ha già inviato una candidatura per quella specifica offerta e, in caso affermativo, il suo stato attuale (*pending*, *accepted*, *withdrawn*).

3. **Verifica Requisiti (`checkCanApply`):** Contatta l'endpoint `/api/v1/students/can-apply/:id` per verificare se l'utente possiede un Profilo di Disponibilità pubblicato e visibile. Senza di esso, il sistema blocca lato client la possibilità di candidarsi, invitando l'utente a completare il proprio profilo.

Per ottimizzare i tempi di caricamento, queste richieste asincrone vengono parallelizzate sfruttando costrutti Javascript moderni come `Promise.all([])`. Durante l'attesa della risposta multipla dal server, l'interfaccia adotta un *Skeleton Loader* strutturato che ricalca l'impaginazione finale della pagina.

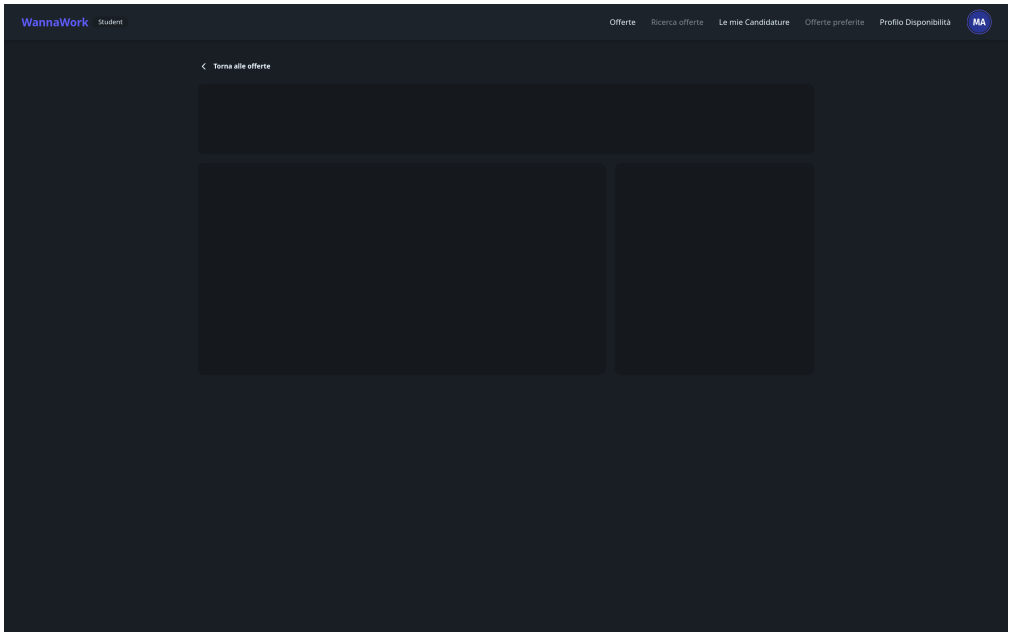

Figura 3.14: *Stato di caricamento asincrono della pagina di dettaglio (Skeleton Loader).*

Il layout è diviso in due colonne logiche (su schermi desktop):

- La colonna principale ospita la descrizione testuale estesa e i badge (dinamici) delle competenze richieste.
- La barra laterale di destra (*sticky sidebar*) funge da riassunto operativo, mostrando icone intuitive per salario, orario e contratto, e integra un *Avatar* aziendale generato al volo in base alla ragione sociale tramite API esterne (`ui-avatars.com`).

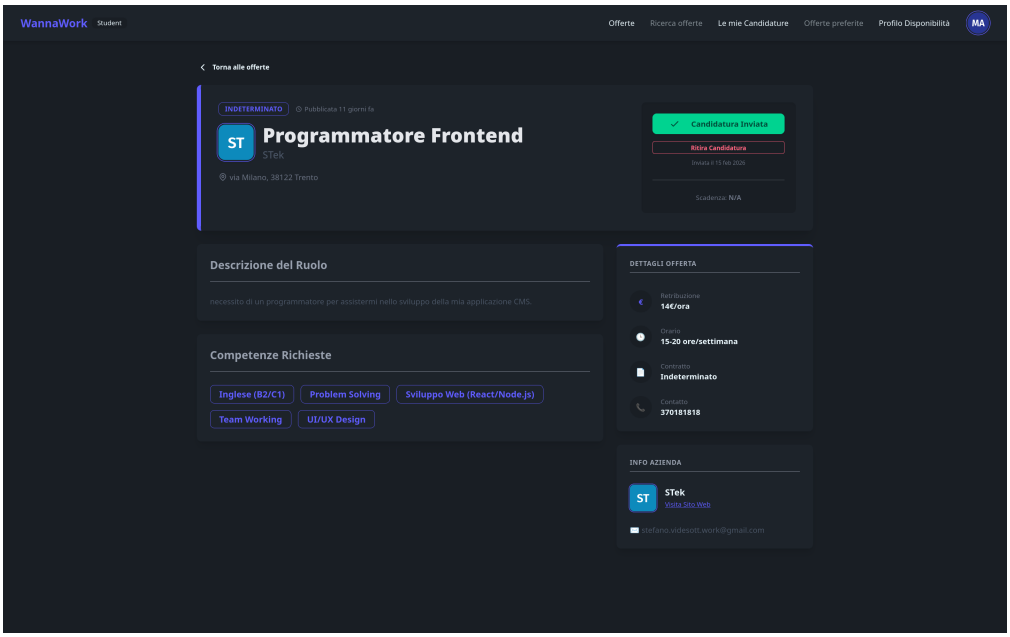

Figura 3.15: *Pagina di dettaglio dell'offerta con layout a due colonne e pulsante di azione dinamico.*

Gestione delle Candidature (Modali) L'azione di candidatura (o del suo ritiro) è protetta dall'invio accidentale tramite l'utilizzo di *Modali* nativi HTML5 (`<dialog>`), stilizzati tramite DaisyUI. Quando l'utente clicca su "Candidati Ora" o "Ritira Candidatura", il modale si sovrappone all'interfaccia bloccando l'interazione sottostante (*modal-backdrop*). Durante l'invio effettivo della richiesta (`POST` o `PATCH`), il pulsante cambia stato, inibendo click multipli fino al ricevimento della risposta dal server, momento in cui la vista aggiorna reattivamente il proprio stato senza necessità di ricaricare la pagina.

3.3.4 Profilo Disponibilità (`AvailabilityProfile`)

Il Profilo di Disponibilità rappresenta il curriculum vitae digitale dello studente all'interno della piattaforma WannaWork. È il prerequisito fondamentale per poter inviare candidature ed essere valutati dalle aziende.

Il componente gestisce tre stati logici ben distinti all'interno della medesima vista, riducendo la proliferazione di file e migliorando la coesione del codice:

1. **Stato Vuoto (Empty State):** Se la chiamata API iniziale (`GET /api/v1/availabilityProfile/me`) restituisce un errore 404 (Profilo non trovato), l'interfaccia mostra una *Hero Section* che invita l'utente a creare il proprio profilo per "iniziare a farsi notare".

2. **Stato di Visualizzazione:** Se il profilo esiste, viene mostrato sotto forma di cruscotto riassuntivo (Dashboard-style), utilizzando le classi `stats` e `badge` di DaisyUI per evidenziare visivamente le competenze, le ore di disponibilità e le esperienze pregresse.

3. **Stato di Modifica/Creazione:** Un modulo interattivo che permette l'inserimento o l'aggiornamento dei dati. Questo stato riutilizza la medesima struttura dati (`form.value`) per entrambe le operazioni, adattando unicamente il metodo HTTP finale (`POST` o `PUT`).

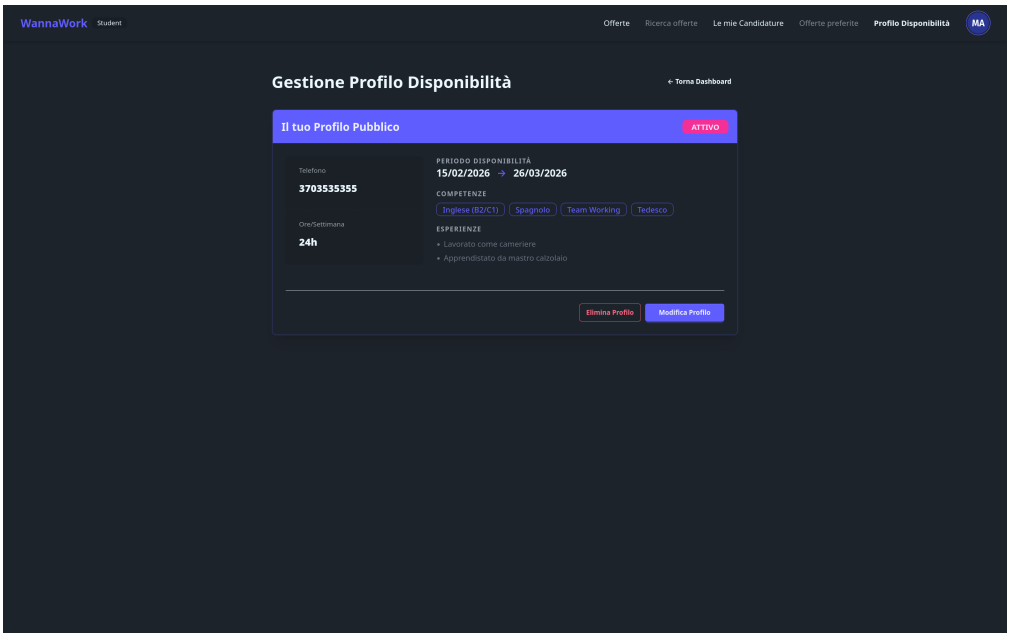

Figura 3.16: *Stato di visualizzazione del Profilo Pubblico attivo.*

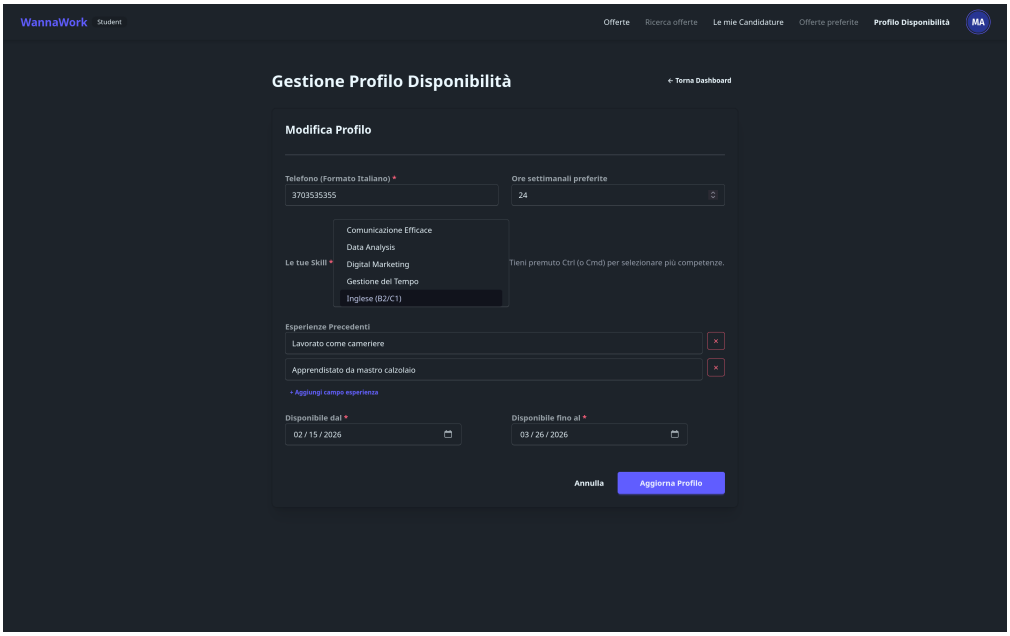

Figura 3.17: *Stato di modifica del profilo con controlli di validazione (es. formato telefono e congruenza date).*

Prevenzione Perdita Dati (Unsaved Changes) Una delle criticità maggiori nei form complessi è la perdita accidentale dei dati compilati dovuta alla navigazione errata o alla chiusura del browser. Per risolvere questo problema, il componente implementa una duplice barriera di sicurezza:

- **Guardia di Navigazione (Vue Router):** Tramite l'hook `onBeforeRouteLeave`, il sistema intercetta qualsiasi tentativo di navigazione interna verso altre viste dell'app. Se rileva modifiche non salvate (confrontando l'oggetto `form` attuale con un backup JSON generato all'apertura del form), blocca la transizione e lancia un `confirm()` nativo per chiedere conferma all'utente.
- **Intercettazione Browser (Window Event):** Poiché il router di Vue non può intercettare la chiusura della scheda o il refresh manuale della pagina (F5), il componente registra un *EventListener* sull'evento `beforeunload` al momento del montaggio (`onMounted`), istruendo il browser a mostrare l'avviso standard di "Modifiche non salvate". Il *listener* viene poi rimosso responsabilmente in fase di smontaggio (`onBeforeUnmount`) per prevenire perdite di memoria (Memory Leaks).

Eliminazione Sicura Considerando la criticità dell'operazione, che comporta il ritiro a catena di tutte le candidature attive dello studente (come definito nelle API backend), l'eliminazione del profilo non avviene con un singolo click. L'azione apre un Modale di Sicurezza che richiede all'utente due conferme esplicite e contestuali: la spunta su una casella di *acknowledgement* (presa visione) e l'inserimento manuale della propria password corrente di accesso al sistema. Se i due requisiti non sono soddisfatti, il pulsante rosso di eliminazione rimane disabilitato.

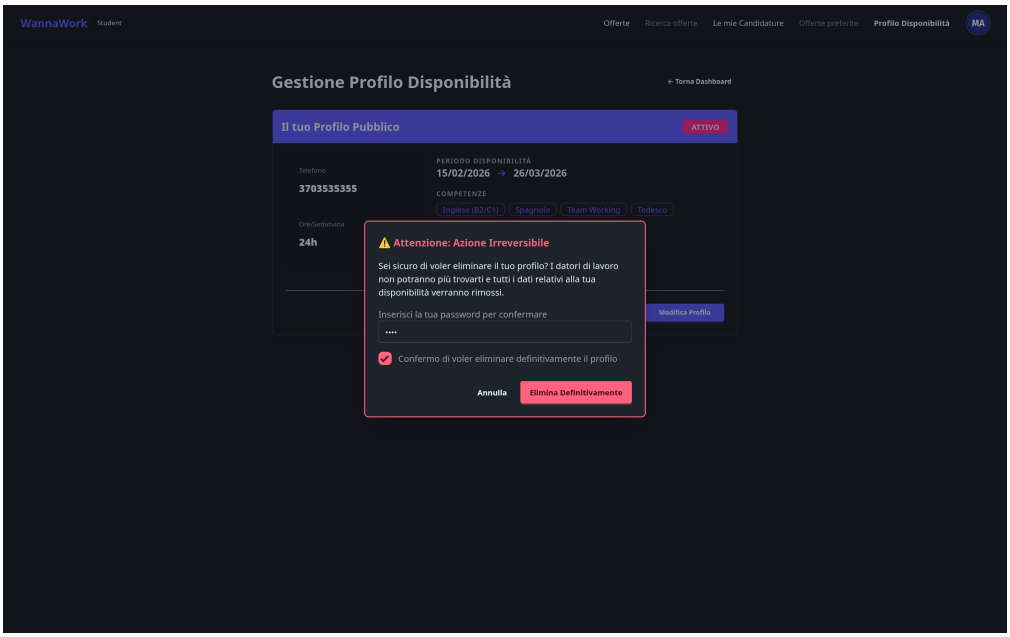

Figura 3.18: *Modale di sicurezza per l'eliminazione del profilo protetto da password.*

3.3.5 Le mie Candidature (`ApplicationsList`) Questa vista fornisce allo studente un riepilogo tabellare (*List View*) di tutte le interazioni intraprese con le aziende presenti sulla piattaforma.

Al momento dell'accesso, l'applicazione interroga l'endpoint protetto `GET /api/v1/applications/student`, il quale restituisce lo storico delle candidature popolato con i dati aggregati dell'offerta e dell'azienda (*companyName*). Qualora non siano presenti candidature, l'interfaccia adatta il proprio stato mostrando un *Empty State* amichevole, costituito da un avviso blu (`alert-info`) contenente una *Call to Action* che invita l'utente a tornare alla Dashboard per esplorare nuove opportunità.

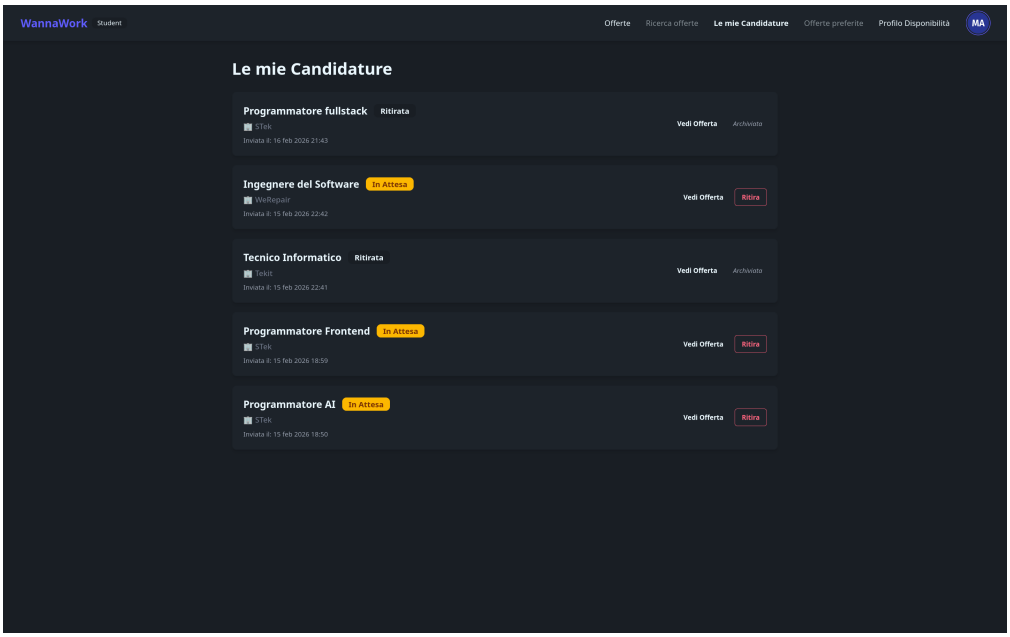

Figura 3.19: *Elenco delle candidature dello studente con indicatori visivi di stato.*

Il design di ogni riga dell'elenco è incentrato sull'immediata comprensibilità dello stato di avanzamento della pratica:

- **Status Badge Dinamici:** Attraverso le funzioni *helper* `getStatusBadge` e `getStatusLabel`, lo stato grezzo proveniente dal database (*pending*, *reviewed*, *accepted*, *rejected*, *withdrawn*) viene tradotto in italiano e associato dinamicamente a un colore semantico di DaisyUI (es. giallo per l'attesa, verde per l'accettazione, grigio per il ritiro).
- **Azioni Contestuali:** A seconda dello stato della candidatura, il sistema espone pulsanti di azione differenti. È sempre possibile navigare verso il dettaglio dell'offerta (`OfferDetail`), ma il pulsante rosso "Ritira" compare **solo** se la candidatura è ancora in fase di valutazione iniziale (*pending* o *reviewed*). Le candidature già esaminate dall'azienda o archiviate non possono essere alterate e mostrano l'etichetta "Archiviata".

Ritiro Volontario della Candidatura Come precedentemente introdotto, il click sul pulsante "Ritira" non esegue immediatamente la chiamata al server. L'azione solleva un Modale di avviso rosso (`border-error`), che riepiloga all'utente il nome della posizione e dell'azienda, avvertendo dell'irreversibilità dell'azione.

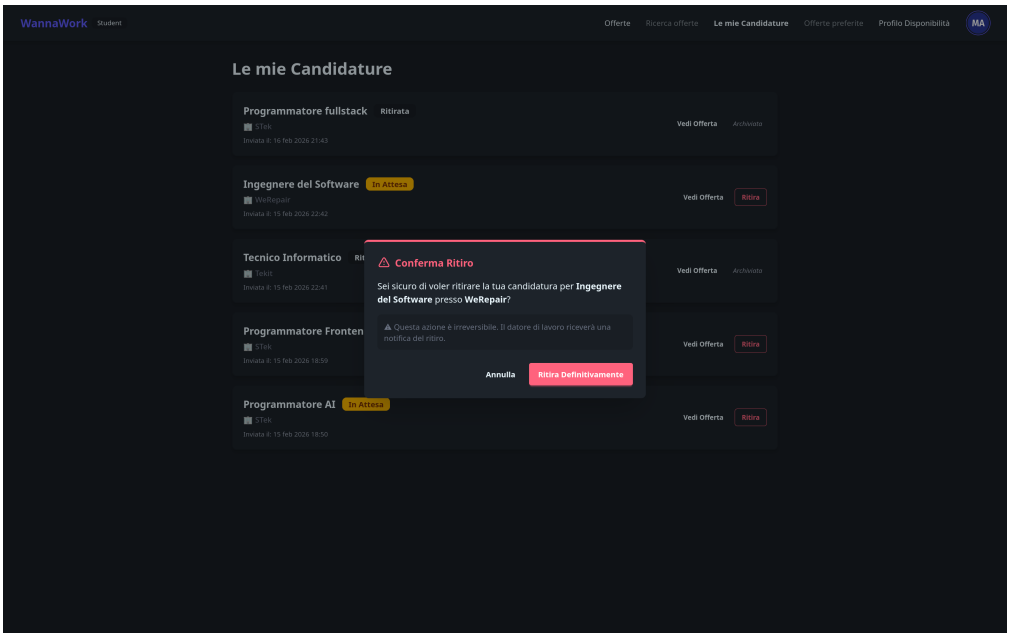

Figura 3.20: *Modale di conferma per il ritiro definitivo di una candidatura in corso.*

L'implementazione logica del ritiro (`confirmWithdraw`) dimostra un'efficiente gestione dello stato reattivo in Vue: a seguito della risposta positiva dal server (`PATCH /api/v1/applications/offer/:id/withdraw`), il componente aggiorna localmente lo stato dell'elemento specifico all'interno dell'array `applications.value`, trasformando istantaneamente il badge in "Ritirata" senza costringere l'applicazione a eseguire una nuova costosa chiamata API (Data Fetching) per ricaricare l'intera lista.

3.4 Area Datore di Lavoro

L'Area Datore di Lavoro (Employer) è l'ambiente riservato alle aziende per la pubblicazione degli annunci di lavoro e la gestione del processo di selezione (recruiting). L'accesso a quest'area è rigidamente protetto dal sistema RBAC e inibito agli utenti con profilo "Studente".

3.4.1 Barra di Navigazione (`EmployerNavbar`) In modo analogo all'area Studenti, la navigazione aziendale è standardizzata da una barra dedicata (`EmployerNavbar`) ancorata in alto. Il design si distingue per l'uso del colore secondario del tema (`text-secondary`), conferendo all'area aziendale un'identità visiva distinta pur mantenendo la coerenza strutturale del framework TailwindCSS.

Il componente espone il link diretto alla consultazione delle "Mie Offerte". Inoltre, per rispettare i vincoli definiti in fase di analisi dei requisiti, la barra ospita il pulsante **Candidati preferiti**; trattandosi di una funzionalità la cui implementazione è posticipata, il link è reso inattivo e opa-

cizzato visivamente, integrando un avviso al passaggio del mouse che ne motiva la disabilitazione temporanea.

All'interno del *dropdown* dell'Avatar aziendale (generato dinamicamente in base alla Ragione Sociale), il sistema fornisce un menu condensato per la navigazione da dispositivi *mobile*, affiancato dalle opzioni per la creazione rapida di una nuova offerta e per il logout di sicurezza.

Figura 3.21: *Barra di navigazione dell'Area Aziendale, caratterizzata dal badge "Business" e dal link WIP inibito.*

3.4.2 Dashboard Aziendale (`EmployerDashboard`) La Dashboard rappresenta il pannello di controllo centrale per le attività di *recruiting*. Al momento del caricamento, la vista interroga l'endpoint protetto `GET /api/v1/offers/my-offers`, il quale restituisce non solo i metadati dell'offerta, ma anche il conteggio aggiornato delle candidature attive per ciascuna di esse (`applicationCount`).

A differenza della visualizzazione a griglia (*Card grid*) utilizzata dagli studenti, la Dashboard Aziendale organizza i dati in un layout tabellare (*Data Table*). Questa scelta UX è mirata a massimizzare la densità di informazioni visibili a colpo d'occhio, facilitando la comparazione e la gestione massiva tipica dei reparti HR.

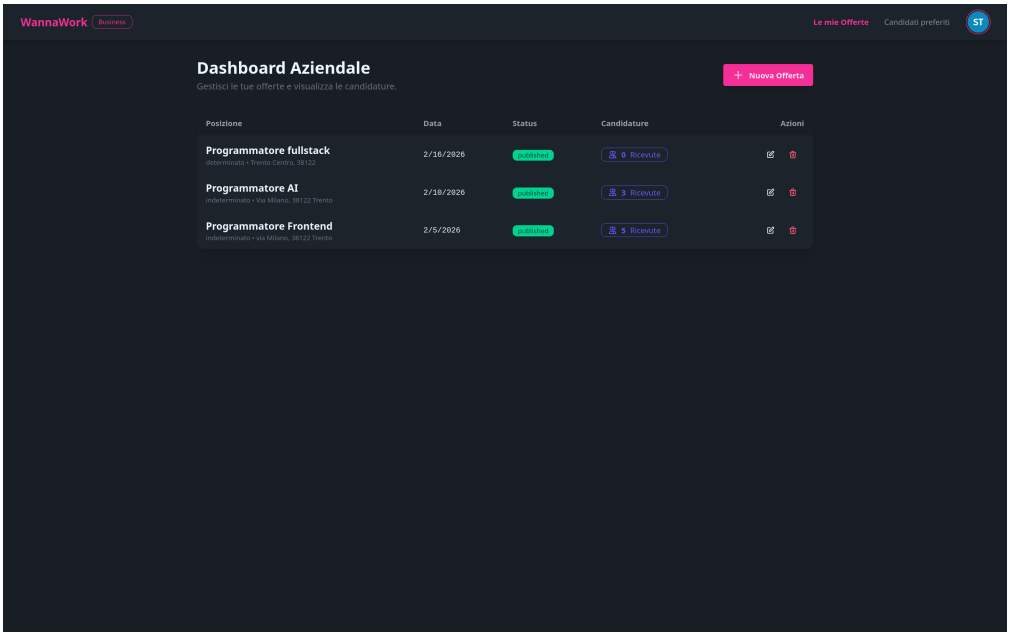

Figura 3.22: *La Dashboard Aziendale con l'elenco tabellare degli annunci pubblicati.*

Ogni riga della tabella espone:

- I dettagli primari dell'offerta (posizione, tipologia contrattuale, sede).
- Uno status visivo dinamico (`getStatusBadge`), che colora l'etichetta in verde (Pubblicata), grigio (Scaduta) o giallo (Bozza/Altro).
- Un **Badge Interattivo delle Candidature**: questo elemento mostra il numero esatto di risposte ricevute. Cliccandolo, il datore viene reindirizzato alla pagina di approfondimento contenente l'elenco dei candidati (`OfferCandidates`).
- Le azioni contestuali di Modifica ed Eliminazione.

Chiusura dell'Annuncio (Eliminazione) L'eliminazione di un'offerta non è un'azione silente. Poiché l'operazione comporta l'annullamento in cascata di tutte le candidature associate (come implementato nel backend), il frontend impone l'apertura di un Modale di sicurezza. L'interfaccia obbliga il datore di lavoro a fornire una **Motivazione** dal menu a tendina (es. "Posizione coperta" o "Offerta scaduta"). Questo dato verrà allegato al payload della richiesta `DELETE` e inviato via email a tutti gli studenti precedentemente candidati, garantendo un processo di selezione trasparente.

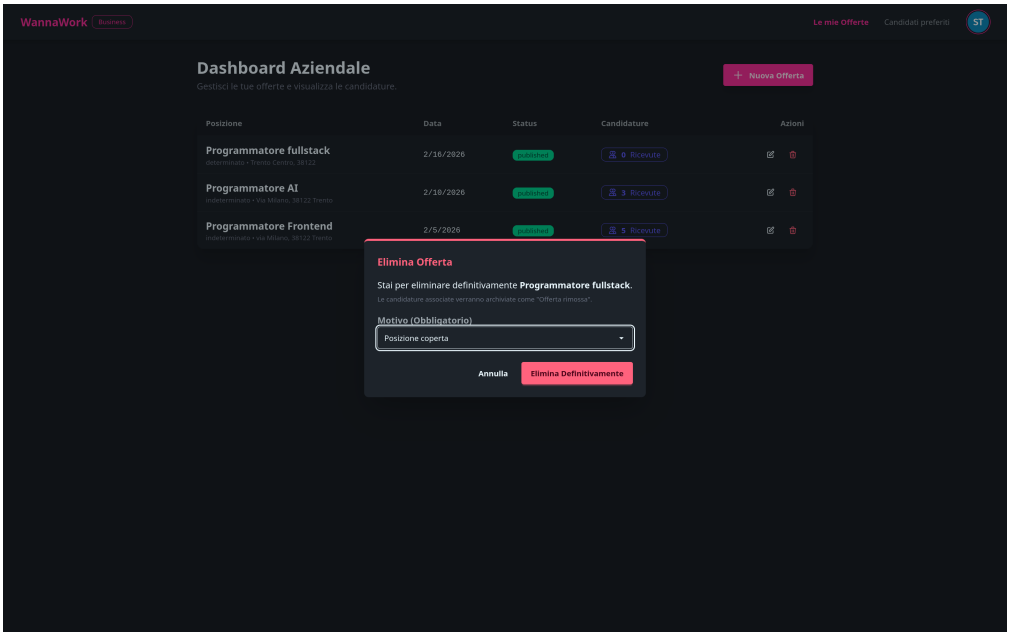

Figura 3.23: *Il Modale obbligatorio per specificare il motivo di rimozione di un annuncio.*

3.4.3 Pubblicazione Offerta (`CreateOffer`)

Questo componente fornisce al Datore di Lavoro il modulo per la creazione e la pubblicazione di un nuovo annuncio lavorativo. Strutturalmente, l'interfaccia adotta un layout a griglia asimmetrica (`lg:grid-cols-3`), riservando la parte principale allo *scrolling* verticale dei form e ancorando sulla destra un pannello di controllo interattivo (*sticky sidebar*).

Il form di inserimento è suddiviso logico-visivamente in tre schede (Cards): Informazioni Base (Titolo, Descrizione, Luogo), Competenze Richieste (caricate dinamicamente dal database) e Condizioni (Orario, Retribuzione, Tipologia di Contratto e Metodi di contatto).

Dal punto di vista della programmazione frontend, il componente fa un uso intensivo delle *Computed Properties* di Vue per fornire un feedback istantaneo all'utente:

- **Validazione Istantanea:** Le regole di validazione (es. `isTitleValid` che richiede da 10 a 100 caratteri, o `isDescriptionValid` che ne richiede almeno 50) sono calcolate in tempo reale a ogni battitura dell'utente. I campi reagiscono colorandosi di rosso in caso di errore, accompagnati da un contatore di caratteri aggiornato live.
- **Checklist di Completamento:** Il pannello laterale utilizza queste *computed properties* per spuntare progressivamente una lista di controllo (*Pronto per pubblicare?*). Il pulsante finale di sottomissione ("Pubblica Ora") rimane disabilitato a livello client finché l'oggetto derivato `isFormValid` non restituisce esito positivo, prevenendo così chiamate API fallimentari.

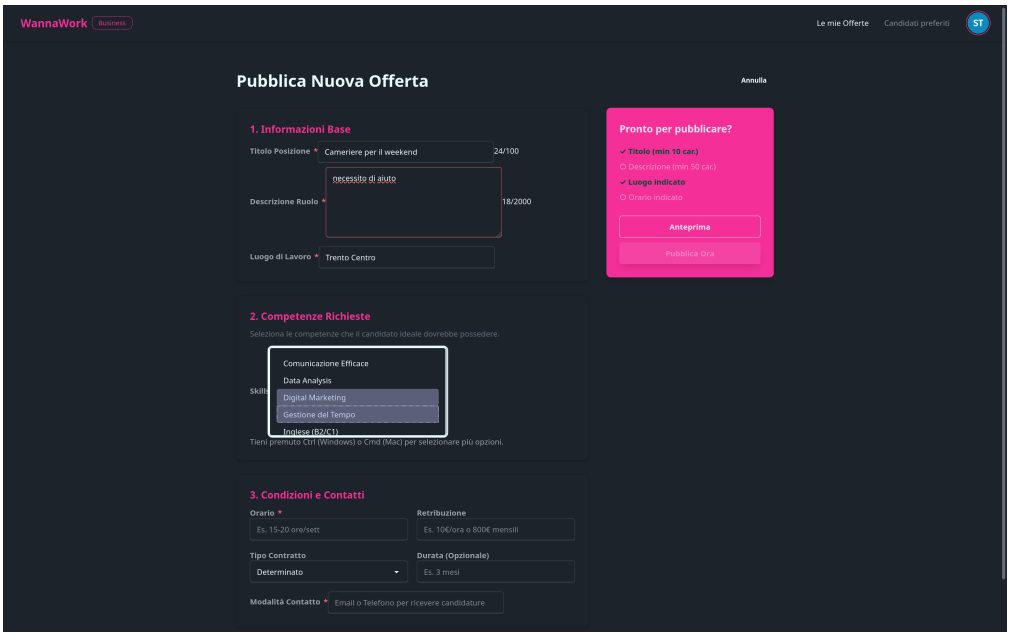

Figura 3.24: *Interfaccia di creazione offerta con il pannello laterale di checklist dinamica.*

Anteprima dell'Annuncio (Preview Mode) Per garantire la massima fedeltà del dato inserito, è stata implementata una funzione di Anteprima (*Live Preview*). Cliccando sull'apposito pulsante nel pannello laterale, si attiva un Modale a pieno schermo (`showPreview = true`) che renderizza istantaneamente i valori attuali dell'oggetto `form.value` applicando la medesima formattazione e gerarchia visiva che gli Studenti vedranno nella loro Dashboard. Questo strumento permette

all'azienda di verificare la leggibilità della descrizione (`whitespace-pre-line`) e la resa visiva dei badge delle competenze (`selectedSkillNames`) prima del salvataggio definitivo sul database.

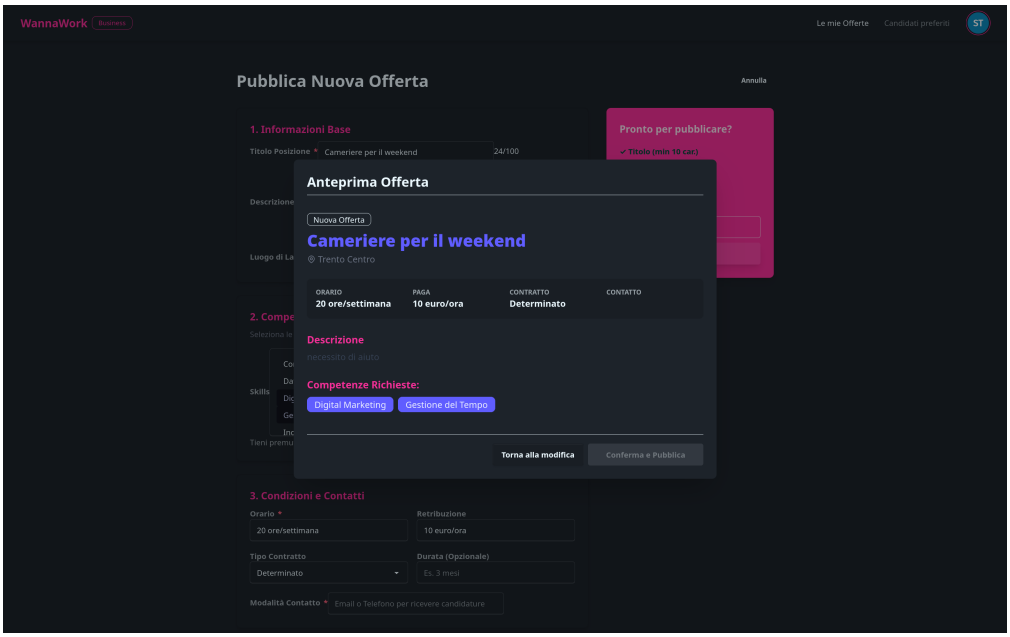

Figura 3.25: *Modale di anteprima dal vivo dell'annuncio prima della pubblicazione definitiva.*

3.4.4 Modifica Offerta (`EditOffer`) Questa vista consente al Datore di Lavoro di aggiornare un annuncio lavorativo precedentemente pubblicato. L'architettura dell'interfaccia e la logica di validazione (*computed properties*) rispecchiano quelle della fase di creazione (`CreateOffer`), garantendo familiarità e coerenza nell'esperienza d'uso (*User Experience*).

L'aspetto tecnologicamente più rilevante di questo componente riguarda la gestione del ciclo di vita dei dati e la prevenzione della sovrascrittura accidentale:

- **Data Fetching Iniziale:** Al momento del montaggio (`onMounted`), il componente esegue una richiesta `GET` parametrizzata (`/api/v1/offers/:id`) per recuperare i dati correnti dell'offerta dal database e popolare i campi del form. A questo punto, lo stato viene "fotografato" e memorizzato come stringa JSON (`originalData`).

- **Gestione del Dirty State:** La *computed property* `hasUnsavedChanges` confronta in tempo reale l'input dell'utente con la fotografia iniziale. Solo se viene rilevata una discrepanza logica, l'interfaccia si anima mostrando un badge di avviso (`animate-pulse`) e abilitando il pulsante "Salva Modifiche". Se l'utente riporta manualmente i campi ai valori di partenza, il pulsante si disabilita nuovamente, risparmiando una chiamata inutile al server (`PUT`).

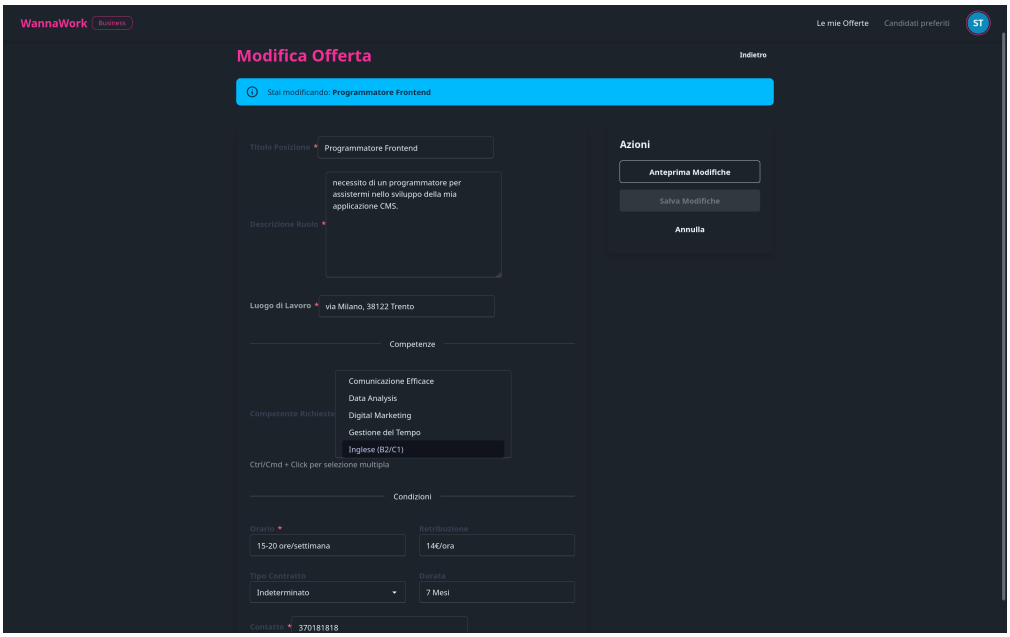

Figura 3.26: *Interfaccia di Modifica Offerta con badge dinamico di segnalazione modifiche non salvate.*

Per prevenire la perdita accidentale di dati causata da errori di navigazione, il componente implementa gli stessi sistemi di sicurezza visti nel Profilo Studente: la *Navigation Guard* di Vue Router (`onBeforeRouteLeave`) e l'intercettazione dell'evento nativo del browser `beforeunload`. In presenza di modifiche non committate, l'utente sarà forzato a confermare esplicitamente l'abbandono della pagina tramite una finestra di dialogo nativa (Modale di sistema).

Come per la creazione, è disponibile il Modale di Anteprima (*Preview*), il quale in questo caso si adatta per evidenziare che ci si trova in modalità "In Modifica", permettendo di revisionare l'annuncio modificato prima dell'effettivo aggiornamento a database.

3.4.5 Gestione Candidati (`OfferCandidates`) Accessibile cliccando sul badge "Candidature Ricevute" direttamente dalla Dashboard Aziendale, questa vista espone al Datore di Lavoro l'elenco degli studenti che si sono proposti per un determinato annuncio.

Al caricamento del componente, viene estratto l'identificativo dell'offerta dai parametri di rotta (`route.params.id`) ed eseguita una richiesta `GET` all'endpoint `/api/v1/offers/:id/candidates`. L'intestazione della pagina si popola dinamicamente con il titolo della posizione ricercata (`offerDetails?.position`), fornendo all'utente un chiaro contesto di navigazione.

Qualora l'annuncio non abbia ancora generato interesse, il componente adotta il consolidato pattern UI dell'*Empty State*, mostrando un box illustrato che invita a tornare successivamente, evitando così di presentare strutture tabellari vuote.

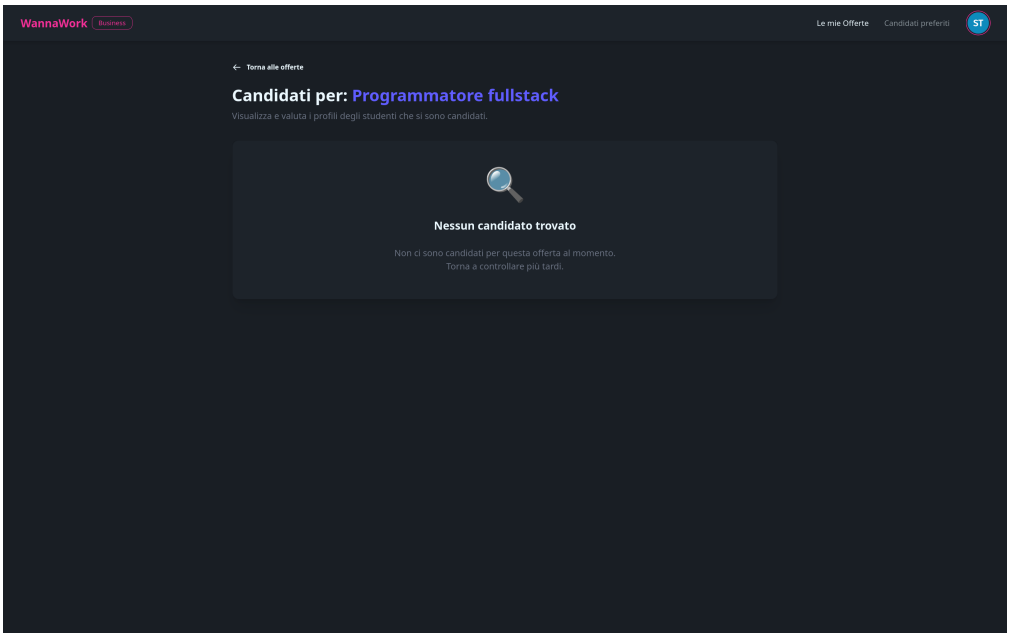

Figura 3.27: *Stato vuoto della pagina di gestione candidati per un'offerta priva di interazioni.*

In presenza di candidature, i profili vengono disposti in una tabella responsive. Il design della tabella è stato ottimizzato per facilitare lo *screening* rapido (valutazione preliminare) da parte dei referenti aziendali:

- **Identificazione Immediata:** Ogni riga include la Data di Candidatura formattata per esteso e l'Avatar dello studente, generato a runtime unendo Nome e Cognome.
- **Smart Tagging delle Competenze:** Per prevenire la rottura del layout (*layout breaking*) in presenza di candidati con innumerevoli competenze, la vista applica un filtro logico di rendering. Vengono stampati a video solo i primi tre badge delle skills (`slice(0, 3)`); i rimanenti vengono raggruppati in un indicatore numerico compatto (es. "+2"), garantendo pulizia e simmetria alle righe della tabella.

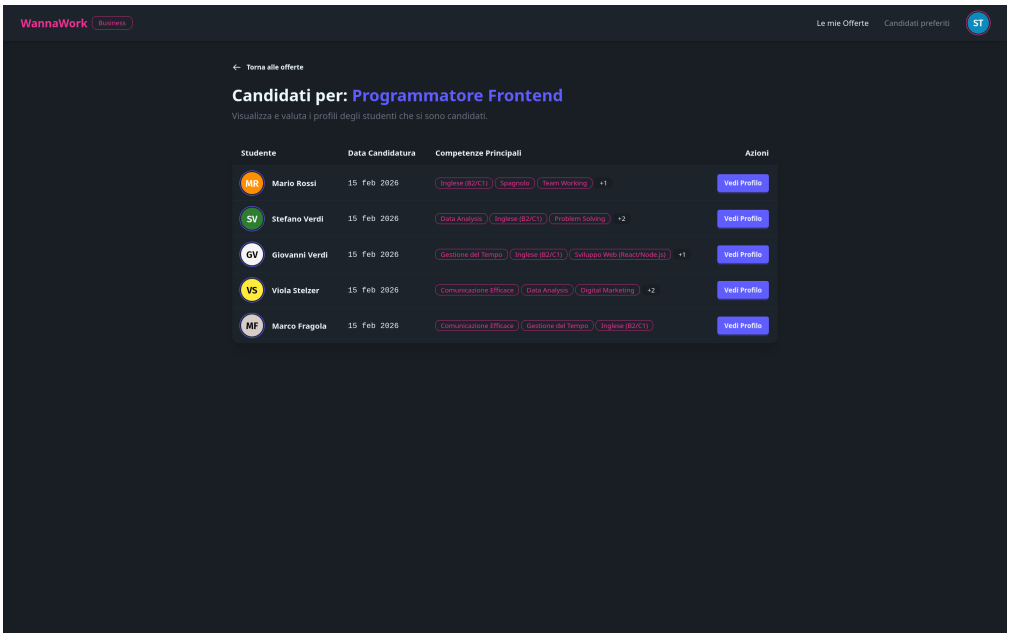

Figura 3.28: *Tabella dei candidati con smart tagging delle competenze per preservare l'ordine visivo.*

L'azione terminale di questa pagina è il pulsante "Vedi Profilo", che estrapola il `profileId` associato allo studente e innesca la navigazione verso la vista di dettaglio del candidato.

3.4.6 Profilo Candidato (`CandidateProfile`) Questa vista chiude il flusso del *recruiting* aziendale, fungendo da Curriculum Vitae digitale dello studente. È accessibile esclusivamente dal datore di lavoro selezionando un candidato dalla lista precedente.

La pagina è concepita per fornire tutte le informazioni necessarie per valutare e contattare il candidato, organizzandole in un layout moderno e di facile lettura. L'intestazione (*Header*) utilizza un gradiente visivo (`bg-gradient-to-r`) e ospita i Dati Personali Sensibili (Nome, Cognome, Email, Numero di Telefono). Si ricorda che tali dati, per motivi di Privacy, non sono pubblici nel database ma diventano accessibili all'azienda **solo nel momento in cui lo studente invia volontariamente la candidatura**.

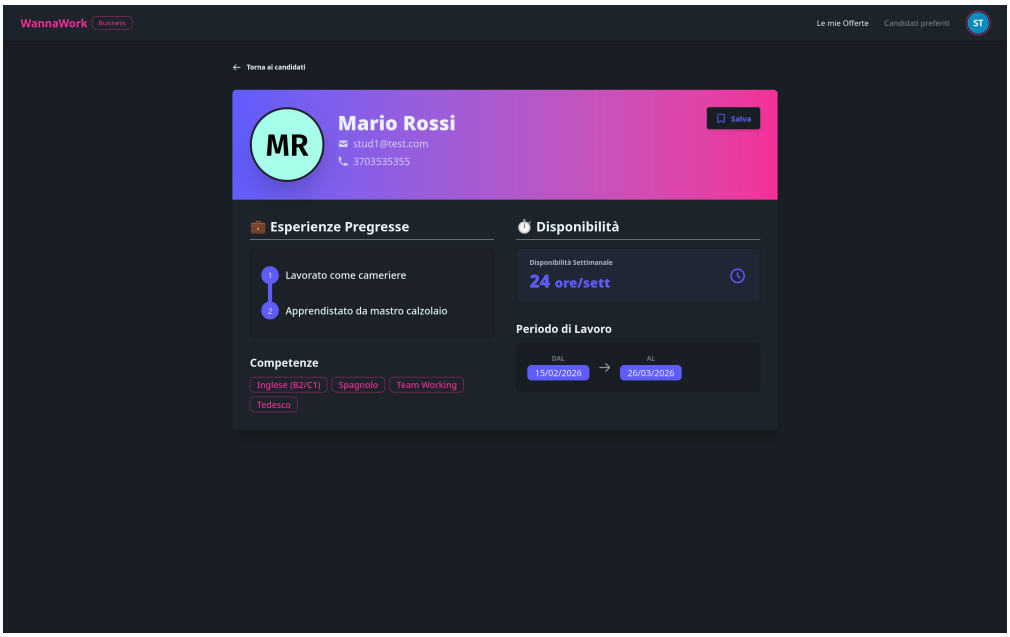

Figura 3.29: *Visualizzazione lato Datore di Lavoro del Profilo di Disponibilità dello studente.*

Il corpo della pagina divide le competenze in due sezioni logiche:

- **Esperienze e Competenze (Sinistra):** Le esperienze pregresse vengono renderizzate utilizzando il componente `steps-vertical` di DaisyUI, che ricrea visivamente l'aspetto di una *Timeline* professionale. Le *Skills* sono presentate immediatamente sotto tramite badge evidenziati.
- **Disponibilità (Destra):** Sfruttando il componente `stats`, le ore settimanali desiderate e l'intervallo temporale di disponibilità (Dal/Al) vengono messi in grande risalto, permettendo all'azienda di verificare istantaneamente se le tempistiche del candidato collimano con le necessità dell'annuncio lavorativo.

Gestione Profili Non Trovati Poiché l'architettura del sistema prevede che uno studente possa ritirare la candidatura o addirittura eliminare definitivamente il proprio profilo in qualsiasi momento, il frontend deve gestire con garbo la possibilità di incappare in un riferimento "orfano". Qualora la chiamata `GET /api/v1/availabilityProfile/:id` restituisca un errore 404, il componente intercetta l'eccezione (`handleNotFound`) e trasforma la pagina in un avviso di errore amichevole. Contestualmente, avvia un conto alla rovescia di 3 secondi (`countdown`), al termine del quale reindirizza automaticamente (`router.back()`) il datore di lavoro alla lista dei candidati ancora attivi.

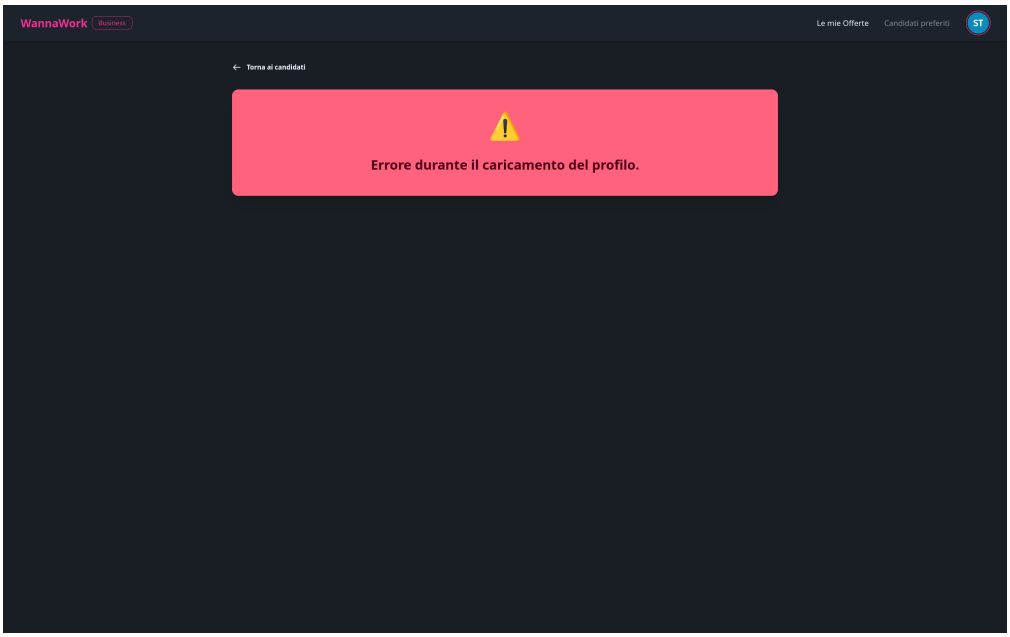

Figura 3.30: *Gestione elegante dell'errore (con fallback automatico) nel caso in cui il candidato abbia eliminato il proprio profilo.*

4. Deployment

Il sistema *WannaWork* adotta un'architettura a microservizi disaccoppiata, in cui il FrontEnd (Vue.js) e il BackEnd (Node.js/Express) sono ospitati ed eseguiti come istanze separate. Per garantire la massima resilienza e dimostrare competenza in diversi scenari di rilascio, il progetto è stato distribuito su una doppia infrastruttura: una piattaforma gestita (**PaaS**) per la continuità operativa e un server proprietario (**Self-Hosted**) per simulare un ambiente di produzione aziendale *on-premise*.

4.1 Infrastruttura Cloud e Servizi Terzi

L'ambiente di hosting principale per entrambi i microservizi è **Render.com**. Per garantire un accesso professionale e centralizzato, le istanze sono state configurate con record CNAME su un dominio personalizzato gestito tramite Cloudflare.

- **FrontEnd (Interfaccia Utente):**
  - Dominio Principale: https://wannawork-frontend.stefanovidesott.com
  - Dominio di Fallback: https://wannawork-frontend.onrender.com
- **BackEnd (API & Database Access):**
  - Dominio Principale: https://wannawork-app.stefanovidesott.com
  - Dominio di Fallback: https://wannawork-app.onrender.com

Il sistema si appoggia inoltre a servizi esterni specializzati per garantire persistenza e comunicazioni:

- **Database:** Il database di produzione è ospitato su un cluster **MongoDB Atlas**, che garantisce alta disponibilità, cifratura dei dati a riposo e backup automatici.
- **Servizio Mail:** Per la gestione delle notifiche e delle comunicazioni transazionali, il sistema utilizza il provider **Brevo** (tramite `smtp-relay.brevo.com`).

4.2 Configurazione delle Variabili d'Ambiente (.env)

Nel rispetto dei principi della metodologia *Twelve-Factor App*, il progetto separa rigorosamente il codice sorgente dalle configurazioni specifiche dell'ambiente di esecuzione. Tutte le credenziali,

gli indirizzi dei servizi esterni e le chiavi crittografiche sono gestite tramite Variabili d'Ambiente (file `.env`), le quali vengono iniettate in fase di esecuzione sia negli ambienti di sviluppo locale sia sulle piattaforme di produzione (Render e Arch Linux).

4.2.1 Configurazione BackEnd Il server Node.js richiede l'impostazione delle seguenti variabili per garantire il corretto avvio e la connessione ai servizi di terze parti:

PORT La porta su cui il server Express resta in ascolto (es. `8080`). In produzione su Render, questo valore viene sovrascritto automaticamente dalla piattaforma.

NODE_ENV Definisce l'ambiente corrente (`development` o `production`). In base a questo valore, il server ottimizza il logging e la gestione degli errori.

MONGODB_URI La stringa di connessione al database. Include le credenziali di accesso al cluster MongoDB Atlas (es. `mongodb+srv://<user>:<password>@cluster...`).

FRONTEND_URL L'indirizzo esatto del client (es. `http://localhost:5173` in locale o il dominio di produzione). Viene utilizzato per configurare i permessi CORS (Cross-Origin Resource Sharing) e per generare i link di reindirizzamento nelle email.

Sicurezza e Autenticazione (JWT):

JWT_SECRET Chiave crittografica complessa utilizzata dall'algoritmo HMAC SHA-256 per firmare e verificare i token di sessione degli utenti.

EMAIL_SECRET Chiave crittografica separata utilizzata per firmare esclusivamente i token temporanei di verifica dell'indirizzo email (scadenza 24h).

Servizio di Posta (SMTP Brevo):

EMAIL_HOST e EMAIL_PORT Endpoint e porta del server SMTP per l'invio delle email transazionali (es. `smtp-relay.brevo.com` sulla porta `587`).

EMAIL_USER e EMAIL_PASS Credenziali di autenticazione fornite dal provider Brevo.

EMAIL_FROM L'indirizzo mittente predefinito da cui gli utenti ricevono le comunicazioni della piattaforma (es. `no-reply@wannawork.com`).

4.2.2 Configurazione FrontEnd A differenza del backend, il frontend costruito con **Vite** richiede una configurazione più snella. Affinché Vite esponga le variabili al codice sorgente Vue.js (tramite `import.meta.env`), queste devono obbligatoriamente iniziare con il prefisso `VITE_`.

`VITE_API_BASE_URL` Definisce la *Base URL* per tutte le chiamate HTTP effettuate tramite Axios. In ambiente di sviluppo punta al backend locale (es. `http://localhost:8080/api/v1`), mentre nei deployment cloud viene impostato sull'URL del server di produzione (es. `https://wannawork-app.onrender.com/api/v1`). In questo modo, l'applicativo client può essere compilato e ridistribuito per ambienti diversi senza alterare una singola riga di codice Javascript.

4.3 Continuous Integration e Continuous Deployment (CI/CD)

L'intero ciclo di rilascio del software è automatizzato tramite **GitHub Actions**. La *pipeline* configurata prevede che, ad ogni `push` o `pull request` verso il branch `main`, venga istanziato un ambiente virtuale per l'installazione delle dipendenze e l'esecuzione dell'intera suite di collaudo (i 62 casi di test unitari e di integrazione documentati nel capitolo relativo all'Implementazione).

Solo se il 100% dei test viene superato con successo (*Passing State*), la pipeline innesca il *web-hook* di Render.com, avviando il processo di *build* e il successivo *deploy* automatico della nuova versione in produzione (*Zero-Downtime Deployment*).

4.4 Deployment Avanzato Self-Hosted (Ambiente di Simulazione Aziendale)

Oltre al deployment standard su piattaforma PaaS (Render.com), è stata realizzata un'infrastruttura di produzione parallela basata su un server reale con sistema operativo **Arch Linux**. Questa scelta mira a simulare un ambiente aziendale *on-premise*, dove il controllo totale dello stack tecnologico permette ottimizzazioni non realizzabili su servizi gestiti.

L'architettura del server è stata progettata seguendo i principi di sicurezza, scalabilità e automazione.

4.4.1 Tecnologie Infrastrutturali Il deployment poggia sui seguenti pilastri tecnologici:

- **Docker & Docker Compose:** Ogni microservizio (FrontEnd e BackEnd) è isolato in un container dedicato. Per il FrontEnd è stato utilizzato un approccio *multi-stage build* per generare artefatti statici ottimizzati serviti tramite un'istanza Nginx interna al container.
- **Nginx come Reverse Proxy:** Il server utilizza Nginx come punto di ingresso unico (*Gateway*). Questo agisce da "vigile urbano", smistando il traffico in arrivo sui sottodomini verso i container Docker corretti.

- **Cloudflare DDNS:** Dato che il server opera su un IP dinamico, è stato implementato un servizio di *Dynamic DNS* dedicato che monitora l'IP pubblico e aggiorna automaticamente i record DNS tramite le API di Cloudflare in tempo reale, mantenendo il servizio sempre raggiungibile.

4.4.2 Sicurezza e Connessione Cifrata A differenza del deployment su Render, la gestione dei certificati è stata configurata e automatizzata manualmente a livello di sistema operativo:

- **Certbot & Let's Encrypt:** È stato configurato un sistema di recupero e rinnovo automatico dei certificati SSL/TLS, garantendo connessioni HTTPS sicure su entrambi i sottodomini.
- **SSL Full (Strict):** Il traffico tra il gateway di Cloudflare e il server Arch Linux è interamente cifrato, impedendo in modo assoluto attacchi di tipo *Man-in-the-Middle*.

4.5 Credenziali di Accesso per il Testing

Al fine di agevolare la valutazione del progetto da parte dei docenti, il database di produzione è stato popolato (tramite *seeding*) con un set di dati fittizi. Questi profili hanno permesso di generare una rete di Offerte, Profili di Disponibilità e Candidature incrociate, rendendo le Dashboard immediatamente ricche di contenuti (come dimostrato negli screenshot di questo documento e nel video di presentazione).

Per accedere all'applicativo ed esplorare i due diversi percorsi basati sul ruolo (RBAC), è possibile utilizzare i seguenti account di test.

Password globale per tutti gli account di test: `testtesttest` *(12 caratteri, nel rispetto dei criteri di sicurezza del sistema).*

Account Studente Questi profili hanno accesso alla bacheca delle offerte, possono creare il proprio profilo di disponibilità e inviare o ritirare candidature.

- `stud1@test.com`
- `stud2@test.com`
- `stud3@test.com`
- `stud4@test.com`
- `stud5@test.com`

Account Datore di Lavoro Questi profili hanno accesso all'area aziendale, possono pubblicare, modificare ed eliminare annunci lavorativi, e visionare i profili di disponibilità degli studenti candidati.

- dat1@gmail.com
- dat2@gmail.com
- dat3@gmail.com
- dat4@gmail.com
- dat5@gmail.com

4.6 Supporto

Per qualsiasi problema tecnico legato all'accesso ai link di deploy, alla visualizzazione dei dati o al login tramite gli account forniti, si prega di contattare l'amministratore del sistema all'indirizzo email: stefano.videsott@studenti.unitn.it.