Politecnico di Torino

Politecnico di Torino

Corso di Laurea

A.a. 2024/2025

Sessione di laurea Dicembre 2025

FacultyApp: integrazione delle API per l'app mobile dei docenti del Politecnico di Torino

Relatore:

Luigi De Russis

Candidato:

Gerardo Maruotti

Ringraziamenti

Desidero innanzitutto esprimere la mia gratitudine al mio relatore Luigi De Russis, ai collaboratori di ISIAD e a tutte le persone che mi hanno accompagnato e supportato lungo l'intero percorso di realizzazione di questa tesi.

Un ringraziamento speciale va ai miei genitori, per il loro sostegno in ogni momento, specialmente nei periodi più difficili, e per avermi permesso di intraprendere questa carriera. Non potrò mai ringraziarvi abbastanza per tutto ciò che avete fatto per me. Ringrazio anche mio fratello per avermi aiutato a rimanere in forma; spero che un giorno si ricordi di includermi nei ringraziamenti della sua tesi.

Ringrazio la mia ragazza per aver sempre creduto in me e per avermi dato la forza di superare tutte le sfide che ho affrontato in questo percorso (e per regalarmi sempre delle belle scarpe). Voglio anche ringraziarla per tutto ciò che ha fatto negli ultimi mesi; non ce l'avrei mai fatta a fare tutto questo senza di lei.

Inoltre, ringrazio tutta la mia famiglia per il supporto ricevuto. In particolare, vorrei ringraziare mia cugina Carmen per avermi fatto scrivere la mia prima riga di codice.

Infine, ringrazio i miei amici e colleghi per avermi supportato (e sopportato) durante tutto questo percorso.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

La crescente digitalizzazione dei processi accademici ha reso le applicazioni *mobile* strumenti essenziali per supportare le attività quotidiane di studenti e personale docente. Al Politecnico di Torino tali esigenze sono soddisfatte attraverso due applicazioni distinte, progettate per rispondere a bisogni operativi tra loro eterogenei. Se la soluzione rivolta agli studenti ha ormai raggiunto un elevato livello di maturità, l'applicazione dedicata ai docenti presenta ancora margini di miglioramento significativi, sia sul piano funzionale sia in termini di usabilità e coerenza con le linee guida grafiche istituzionali.

PoliTO Faculty è il progetto che ha l'obiettivo di colmare questo divario, elevando l'applicazione *mobile* per i docenti agli stessi standard qualitativi raggiunti da quella per gli studenti. Il progetto è nato in collaborazione con ISIAD, che ha supportato attivamente l'intero processo di raccolta dei requisiti, proposta di soluzioni e sviluppo di nuove api per esporre funzionalità, che fino ad ora erano accessibili esclusivamente tramite portali web.

Il punto di partenza di questo lavoro consiste nel comprendere le limitazioni e i problemi dell'attuale applicazione. Nel corso di questa attività sono emerse criticità tipiche dei sistemi *mobile* che crescono in modo stratificato: flussi non sempre allineati ai reali bisogni del personale docente, integrazioni eterogenee con i sistemi di Ateneo e mancanza di funzionalità ritenute essenziali dagli utenti.

1.2 Obiettivi della tesi

L'obiettivo generale di questa tesi è la progettazione e integrazione di un livello API robusto, documentato e verificabile per *PoliTO Faculty*, partendo dall'analisi dei bisogni degli utenti dell'app attuale.

Gli obiettivi specifici sono:

- **Analisi dei bisogni**: definire chiaramente quali sono i bisogni reali del personale docente, tramite un'indagine strutturata.
- **Definizione di requisiti**: creazione di requisiti funzionali e non funzionali che il nuovo sistema dovrà rispettare [1].
- **Selezione delle funzionalità**: tradurre i bisogni emersi in un insieme di funzionalità, collegando ciascun requisito alle operazioni API corrispondenti attraverso una mappatura *requisito* → *endpoint*.
- **Progettazione contrattuale OpenAPI**: definire schemi, parametri e risposte coerenti (formati data/ora, identificativi, nullabilità, codici d'errore), organizzando le specifiche in una struttura che facilita la collaborazione e risulta mantenibile a lungo termine [2].
- **Integrazione delle API con il frontend**: connettere l'interfaccia grafica alle nuove API definite, rendendo l'applicazione pronta per gestire dati utente reali.

Perimetro escluso. Non rientrano nel perimetro di questo lavoro: la riprogettazione visuale delle interfacce e lo sviluppo dell'infrastruttura backend dei sistemi d'Ateneo. Tali aspetti sono considerati vincoli o contesti di integrazione e vengono trattati solo dove necessario a motivare scelte API.

1.3 Struttura del lavoro

La restante parte dell'elaborato è organizzato come segue:

- **Capitolo 2 – Contesto e background**: descrizione dell'ecosistema digitale dell'Ateneo, panoramica dei servizi di riferimento e dei concetti tecnici utilizzati.
- **Capitolo 3 – Analisi dei bisogni e requisiti**: sintesi delle esigenze funzionali e non funzionali che guidano la progettazione delle API.
- **Capitolo 4 – Architettura e progettazione delle API**: struttura informativa, modellazione delle entità, criteri di versionamento, politiche di sicurezza e organizzazione della repository OpenAPI.
- **Capitolo 5 – Implementazione e integrazione**: pipeline di validazione e pubblicazione delle specifiche, generazione dei client e integrazione con il frontend, migrazione da dati *mock* a chiamate reali attraverso adattatori e *hook*.

- **Capitolo 6 – Testing e validazione tecnica**: metodologia di verifica, casi d'uso coperti, metriche di qualità e risultati ottenuti.
- **Capitolo 7 – Conclusioni e sviluppi futuri**: risultati, limiti, opportunità di estensione.

Questa struttura accompagna il lettore dal contesto motivazionale alla formalizzazione dei contratti, fino all'integrazione operativa e alla validazione tecnica dei flussi considerati prioritari per l'utenza docente del Politecnico di Torino.

Capitolo 2

Background

2.1 Ecosistema digitale del Politecnico di Torino

L'ecosistema digitale del Politecnico di Torino comprende un'infrastruttura gestita internamente da ISIAD, il dipartimento IT dell'Ateneo, che si occupa dello sviluppo e della manutenzione dei principali servizi digitali destinati alla comunità accademica. Questa organizzazione rappresenta un punto di forza in termini di controllo e personalizzazione delle tecnologie utilizzate, ma richiede al tempo stesso una distribuzione attenta delle risorse e la definizione di priorità orientate ai progetti di maggiore impatto.

Negli anni, la priorità attribuita allo sviluppo dei servizi rivolti agli studenti ha finito per lasciare in secondo piano le esigenze del personale docente. Di conseguenza, pur disponendo di un solido insieme di servizi accessibili via web, la corrispondente versione mobile ha mostrato limiti evidenti sotto il profilo dell'interfaccia grafica e dell'esperienza d'uso. È infatti emerso che la maggior parte dei docenti tende a privilegiare l'interfaccia web, ritenuta più completa, anche a costo di rinunciare alla semplicità e immediatezza di accesso offerte dalla soluzione mobile.

2.2 La nascita del progetto PoliTO Faculty

La crescente diffusione dei dispositivi *mobili* nell'ambito universitario ha evidenziato che il personale docente e di ricerca presenta esigenze specifiche, differenti da quelle della popolazione studentesca. Tra queste rientrano la gestione delle attività didattiche (insegnamenti, esami), la pianificazione e prenotazione di spazi e risorse, la consultazione di informazioni operative e la gestione dei flussi amministrativi, che comprendono la firma digitale di documenti. In tale contesto nasce *PoliTO Faculty*, un'applicazione progettata per offrire a docenti e ricercatori un accesso

integrato, coerente e ottimizzato ai principali servizi d'Ateneo in mobilità, con particolare attenzione a:

- **Coinvolgimento continuo dei docenti**, applicando la metodologia *user-centered* design (UCD [3]), per progettare un'interfaccia grafica focalizzata sull'utente finale e sulle sue esigenze e contesto di utilizzo.
- **Approccio *API-first***, definendo specifiche OpenAPI a supporto della comunicazione tra client e server, con l'obiettivo di garantire coerenza nell'interfaccia esposta e di stabilire a priori le modalità di utilizzo delle API fornite dai servizi backend esistenti e consentire lo sviluppo tramite mock affidabili [4];
- **Architettura modulare e componibile**, per garantire un'elevata mantenibilità nel tempo e coerenza con il design system istituzionale dell'Ateneo.

2.3 Tecnologie e concetti di riferimento

Il progetto adotta un approccio *API-first* in cui le interfacce HTTP sono descritte con la *OpenAPI Specification* (OAS) 3.1.1, così da abilitare la validazione automatica della specifica e la generazione di client tipizzati [2].

Sul lato applicativo, il frontend *mobile* è basato su *React Native*, un framework che consente di sviluppare interfacce native multi-piattaforma con TypeScript [5, 6].

Capitolo 3

Analisi dei Bisogni e Definizione dei Requisiti

Il seguente capitolo illustra il processo di raccolta e analisi dei dati necessari per la definizione dei requisiti funzionali e non funzionali dell'applicazione *PoliTO Faculty*. I risultati ottenuti hanno costituito la base per individuare le funzionalità previste e l'attribuzione delle relative priorità, determinate in funzione dei riscontri e delle esigenze emerse dal confronto con gli utenti.

3.1 Metodologia di raccolta dati

Per ottenere un riscontro oggettivo sulle modalità di utilizzo e sui limiti dell'applicazione utilizzata attualmente, è stato generato un questionario con l'obiettivo di raccogliere il maggior numero possibile di riscontri e suggerimenti da parte del personale docente e di ricerca. I dati ottenuti sono stati successivamente analizzati per individuare dei requisiti che rispecchiassero i bisogni reali della maggioranza degli utenti.

Il questionario è stato realizzato in collaborazione con il Dipartimento ISIAD, che ha supportato sia la distribuzione attraverso canali ufficiali, sia la successiva analisi dei risultati volta alla definizione dei requisiti.

3.1.1 Partecipanti e profili

L'obiettivo principale del questionario era quello di raccogliere quanti più dati eterogenei possibili. Superata la fase di validazione, il questionario è stato inviato a professori, ricercatori, assegnisti, docenti esterni e collaboratori didattici, in modo da ottenere una solida base di riscontri per definire dei requisiti che prendessero in considerazione le esigenze di tutti i futuri utenti.

Il campione di risposte è composto da **681** risposte, con distribuzione anagrafica bilanciata: dal 1960 al 1999 risulta prevalente la fasce d'età compresa tra il 1970 e il 1979, con un **25,5%** di risposte sul totale. Il tasso di risposta per i dipartimenti è eterogeneo, quindi garantisce sufficiente affidabilità nella definizione delle esigenze comuni.

Per quanto riguarda il ruolo ricoperto dai partecipanti, la distribuzione delle risposte è la seguente:

- Professori: 43%
- Ricercatori: 36%
- Assegnisti: 25%
- Docenti esterni e/o collaboratori didattici: 19%

3.2 Sintesi dei risultati e dei bisogni individuati

3.2.1 Struttura del questionario

Il questionario è stato strutturato in circa trenta domande, suddivise tra quesiti a risposta chiusa e a risposta aperta, e organizzate in modo da coprire i seguenti ambiti tematici:

- Informazioni anagrafiche dei partecipanti;
- Tipologia di dispositivi utilizzati e frequenza d'uso;
- Modalità e preferenze di accesso ai servizi offerti dall'Ateneo da dispositivi mobile;
- Valutazioni sull'esperienza d'uso dell'applicazione attualmente disponibile;
- Proposte di miglioramento e suggerimenti relative a nuove funzionalità per la nuova *PoliTO Faculty*;

Questa organizzazione ha consentito di raccogliere dati facilmente analizzabili e di ottenere indicazioni di notevole valore per la definizione delle principali funzionalità dell'applicazione, nonché per l'attribuzione delle relative priorità.

3.2.2 Analisi dei dati

Per eseguire un'analisi accurata dei dati raccolti, sono stati sviluppati appositi script in *Python* dedicati all'elaborazione delle risposte a scelta chiusa, rappresentando i risultati ottenuti mediante grafici a barre, con l'obiettivo visualizzare in modo chiaro e dettagliato la distribuzione delle risposte. Come mostrato in Figura 3.1, questa rappresentazione consente di individuare con chiarezza le funzionalità più utilizzate dell'applicazione attuale. È quindi evidente che la funzionalità più utilizzata sia quella connessa alla gestione degli appelli degli esami, seguita dalla visualizzazione del calendario e dell'orario delle lezioni.

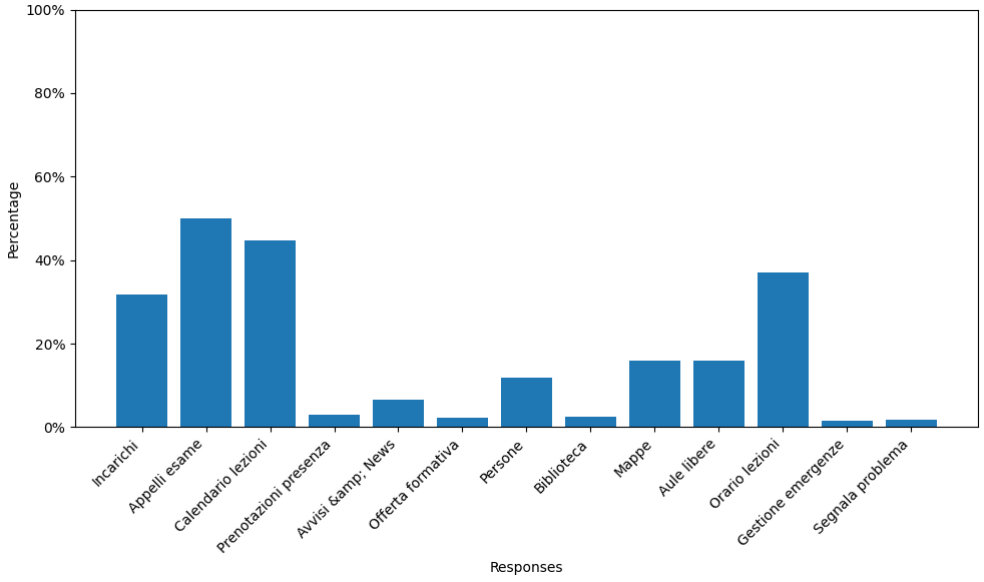

Figura 3.1: Funzionalità esistenti e percentuali di utilizzo

3.2.3 Analisi qualitativa tramite LLM

Per l'analisi delle risposte aperte è stato adottato un approccio supportato da *LLM* (Large Language Models). Si tratta di sistemi statistici addestrati su grandi quantità di testi, in grado di individuare pattern ricorrenti e generare sintesi coerenti a partire da input eterogenei [7].

Le risposte sono state raggruppate per domanda ed elaborate in gruppi, chiedendo al modello di individuare i temi ricorrenti, proporre una prima categorizzazione e sintetizzare i principali bisogni espressi dai docenti.

I risultati prodotti automaticamente non sono stati accolti senza un'adeguata verifica. Per ogni gruppo di risposte è stato selezionato un campione significativo che

è stato confrontato manualmente con le categorie proposte dal modello, correggendo accorpamenti troppo generici, rinominando le etichette poco chiare ed eliminando eventuali temi non supportati dai dati.

Questo processo ha permesso di ridurre i tempi di analisi senza rinunciare a un controllo umano accurato. Rimangono tuttavia alcuni limiti intrinseci all'uso di modelli di linguaggio, come la tendenza a semplificare eccessivamente formulazioni ambigue o a sottorappresentare risposte isolate ma potenzialmente rilevanti, oltre al rischio di introdurre bias nell'annotazione [8]. Per questo motivo, le conclusioni tratte dalle domande aperte sono state considerate come un supporto alle evidenze quantitative, e non come l'unica base per la definizione dei requisiti.

Oltre agli aspetti operativi, l'uso degli LLM per l'analisi delle risposte aperte è stato motivato da considerazioni pratiche legate ai tempi e al volume dei dati raccolti. In alternativa, sarebbe stato possibile procedere con una codifica manuale completa, leggendo e classificando una per una le risposte dei docenti. Questa soluzione avrebbe garantito un controllo molto fine sul processo di categorizzazione, ma avrebbe richiesto uno sforzo significativo e poco sostenibile in questo contesto, rallentando le fasi successive di progettazione.

Sono stati valutati anche strumenti dedicati di *sentiment analysis* o *thematic analysis*, che però risultavano meno flessibili nel gestire contemporaneamente il contenuto testuale e il contesto specifico dell'applicazione. Gli LLM sono stati quindi utilizzati come supporto per ottenere in tempi brevi una prima organizzazione delle risposte in temi ricorrenti, mantenendo comunque un ruolo attivo nella revisione e nell'interpretazione dei risultati. In questo modo l'analisi qualitativa ha potuto combinare la rapidità offerta dai modelli con una lettura critica orientata alla definizione dei requisiti dell'applicazione.

3.2.4 Principali risultati emersi

Il processo di analisi dei dati ha consentito di ottenere una chiara visione sui temi di maggior interesse per gli utenti. Con riferimenti all'attuale applicazione, il **66%** dichiara di utilizzarla, il **22%** non l'ha mai installata, mentre il **12%** riferisce di averla in successivamente disinstallata. Le sezioni maggiormente utilizzate risultano essere appelli, calendario/orario e incarichi, mentre tra e funzionalità richieste risultano mappe e la disponibilità delle aule libere.

L'analisi approfondita dei risultati ha permesso di individuare le seguenti aree tematiche:

- **Miglioramento di funzionalità esistenti**: gli appelli d'esame, il calendario e l'orario delle lezioni sono spesso percepiti come poco intuitivi e complessi da consultare.

- **Introduzione di funzionalità amministrative**: numerosi utenti hanno richiesto di integrare nell'applicazione servizi attualmente disponibili solo tramite portali web, come la prenotazione degli spazi, la firma digitale dei documenti e la gestione delle segnalazioni di guasti o emergenze.
- **Maggiore integrazione con i servizi didattici**: è emersa la necessità di migliorare il collegamento tra l'app e i servizi dedicati alla gestione della didattica.
- **Espandere le modalità di comunicazione e la gestione del materiale didattico**: diversi partecipanti hanno suggerito strumenti più efficaci per la comunicazione con gli studenti e la condivisione del materiale didattico.
- **Estensione delle informazioni del profilo docente**: tra le proposte aggiuntive figura la possibilità di includere, oltre ai dati relativi alla didattica, anche informazioni contrattuali. È stata inoltre evidenziata l'importanza di disporre di una funzione di ricerca sia del personale docente sia degli studenti.

Infine, tra le criticità evidenziate emergono la stabilità del sistema e i tempi di caricamento, l'affidabilità delle notifiche, la scarsa uniformità dell'interfaccia grafica, la mancanza di parità funzionale rispetto ai servizi web e la frammentazione delle integrazioni esistenti.

3.3 Definizione dei requisiti

Una volta individuate le principali esigenze degli utenti tramite analisi quantitative e qualitative, il passo successivo ha riguardato la definizione dei requisiti dell'applicazione.

Sono stati identificati sia requisiti funzionali sia requisiti non funzionali: i primi riguardano ciò che il sistema deve permettere di fare, mentre i secondi descrivono come il sistema deve operare, indicando le proprietà qualitative e i vincoli tecnici legati a prestazioni, sicurezza e usabilità [1].

La combinazione dei bisogni emersi, dei requisiti funzionali e dei requisiti non funzionali costituisce la base per la progettazione delle API. In questo modo l'applicazione può raggiungere, dove necessario, la parità funzionale con i servizi web, migliorandone l'esperienza d'uso.

3.3.1 Requisiti funzionali

L'analisi dei dati ha permesso di definire con precisione le aree tematiche di intervento prioritarie. Come evidenziato nella sintesi dei risultati, è emersa la necessità non solo di migliorare le funzionalità esistenti, come la gestione degli

appelli e degli orari, ma anche di introdurre nuovi servizi amministrativi e di espandere l'integrazione con i servizi didattici. Sono state inoltre considerate le criticità generali relative alla stabilità, ai tempi di caricamento, all'uniformità dell'interfaccia e all'affidabilità delle notifiche.

I requisiti funzionali (FR) sono stati definiti per rispondere in modo mirato a tali esigenze, con l'obiettivo di colmare il divario rispetto ai servizi web e di migliorare in modo significativo l'esperienza d'uso complessiva. La Tabella 3.1 schematizza tali requisiti, evidenziandone la priorità.

3.3.2 Requisiti non funzionali

I requisiti funzionali spiegano cosa deve fare il sistema, mentre quelli non funzionali (NFR) riguardano il modo in cui queste attività devono essere garantite. Si tratta di caratteristiche qualitative e vincoli tecnici che contribuiscono a rendere l'applicazione stabile, reattiva e allineata agli standard previsti.

Considerando questi aspetti, sono stati delineati otto requisiti non funzionali (schematizzati nella Tabella 3.2), che, nel loro insieme, contribuiscono a garantire degli standard di qualità elevati.

L'assegnazione della priorità a ciascun requisito non è stata effettuata in modo arbitrario, ma a partire da una combinazione di informazioni quantitative e qualitative. In primo luogo, per ogni funzionalità è stata considerata la diffusione della richiesta all'interno del questionario, analizzando quante persone avevano indicato esplicitamente quel bisogno o una variante riconducibile allo stesso tema. I requisiti associati a funzionalità fortemente richieste sono stati naturalmente candidati a una priorità più alta.

Accanto a questo aspetto, è stato stimato in maniera qualitativa lo sforzo necessario per l'implementazione, tenendo conto della possibilità di riutilizzare funzionalità già presenti nella PoliTO Students app. I requisiti che potevano essere coperti estendendo flussi esistenti o adattando logiche già implementate sono stati considerati più convenienti da affrontare nelle prime fasi di evoluzione di PoliTO Faculty. Infine, è stata posta particolare attenzione alle motivazioni espresse nelle risposte aperte: alcune funzionalità, pur non essendo le più citate, sono emerse come particolarmente utili per semplificare attività critiche per il personale docente e sono state quindi promosse a priorità più elevata.

Nel complesso, la priorità assegnata riflette un compromesso tra impatto percepito dai docenti, diffusione delle richieste e costo stimato di realizzazione. Questo approccio ha permesso di individuare un nucleo di requisiti principali su cui concentrare il lavoro di progettazione, mantenendo comunque traccia delle funzionalità meno urgenti ma potenzialmente rilevanti per evoluzioni successive dell'applicazione.

ID	Titolo	Descrizione	Priorità
FR1	Gestione utenti	Accesso sicuro all'applicazione e gestione delle informazioni personali.	Alta
FR2	Gestione esami	Consultazione degli appelli, dei relativi dettagli e comunicazioni agli studenti.	Alta
FR3	Incarichi didattici	Gestione dei corsi, collaboratori, materiali e avvisi del docente.	Alta
FR4	Gestione amministrativa	Supporto ad attività amministrative: prenotazioni, firma digitale dei documenti e mappa di ateneo.	Media
FR6	Comunicazione e collaborazione	Accesso ad avvisi, ricerca persone e contenuti informativi dell'Ateneo.	Alta
FR7	Gestione emergenze	Accesso a numeri utili, segnalazioni e procedure di emergenza.	Alta
FR8	Gestione missioni	Inserimento e monitoraggio di trasferte e rimborsi.	Bassa
FR9	Integrazione con servizi esterni	Collegamento con Moodle e altri sistemi esterni istituzionali.	Bassa

Tabella 3.1: Requisiti Funzionali PoliTO Faculty

3.3.3 Dai requisiti alla progettazione delle API

In sintesi, l'analisi dei dati raccolti ha permesso di chiarire quali attività quotidiane i docenti preferirebbero poter gestire da mobile e quali criticità dell'applicazione esistente risultano più rilevanti. A partire da queste evidenze sono stati individuati

ID	Tipo	Descrizione	Priorità
NFR1	Prestazioni	Tempi di caricamento rapidi (inferiori a 3 secondi) e fluidità nella navigazione .	Alta
NFR2	Usabilità	Interfaccia chiara e ottimizzata per dispositivi mobili, coerente con l'app studenti PoliTO [3].	Alta
NFR3	Accessibilità	Conformità alle linee guida WCAG e con le tecnologie assistive [9].	Media
NFR4	Sicurezza	Protezione dei dati personali, autenticazione con credenziali.	Alta
NFR5	Affidabilità	Stabilità, risposta rapida e assenza di crash.	Alta
NFR6	Scalabilità	Supporto ad un elevato numero di utenti simultanei.	Media
NFR7	Compatibilità	Funzionamento stabile su dispositivi Android e iOS.	Alta
NFR8	Coerenza grafica	Uso dei componenti UI ufficiali dell'Ateneo.	Alta

Tabella 3.2: Requisiti Non Funzionali PoliTO Faculty

insiemi di requisiti funzionali e non funzionali che delineano, rispettivamente, i principali ambiti di servizio e le proprietà qualitative che la nuova soluzione dovrà rispettare.

Questi insiemi di requisiti rappresentano il punto di partenza per la fase di progettazione tecnica. Nei capitoli successivi verranno presi come riferimento per identificare i domini funzionali da coprire tramite servizi applicativi dedicati e per definire i contratti delle API che metteranno a disposizione del client mobile le informazioni e le operazioni necessarie. La progettazione delle API non nasce quindi in astratto, ma è radicata nei bisogni emersi dal questionario e nella sintesi strutturata riportata in questo capitolo.

Capitolo 4

Architettura del Sistema e Progettazione delle API

Il presente capitolo illustra il processo di progettazione delle API messe a disposizione di *PoliTO Faculty*. La definizione delle API si è basata sui requisiti funzionali e non funzionali individuati nel capitolo precedente, consentendo di progettare API specificamente calibrate sulle funzionalità previste dall'applicazione e di agevolarne l'integrazione all'interno del codice.

Il lavoro è stato svolto all'interno del progetto *api-spec*, contenente la specifica OpenAPI già utilizzata dall'applicazione PoliTO Students [10]. In questo contesto sono stati affrontati tre aspetti principali:

- la riorganizzazione tecnologica e strutturale della specifica, passando da un file monolitico a una struttura modulare;
- la modellazione delle entità e dei flussi di comunicazione comuni a studenti e docenti;
- la progettazione dei nuovi endpoint REST, con particolare attenzione alla sicurezza, alla coerenza semantica e alla manutenibilità del repository.

4.1 Scelte tecnologiche e pattern architetturali

Il lavoro è partito dall'analisi della specifica `openapi.yaml` esistente, avente standard OpenAPI `3.0.3`. In questa versione la documentazione delle API era organizzata come un unico file monolitico, pensato principalmente per l'applicazione *PoliTO Students*.

L'analisi ha messo in evidenza alcune criticità strutturali:

- **Struttura studenti-centrica**: il catalogo delle operazioni copriva in modo efficace le esigenze della popolazione studentesca (profilo, carriera, lezioni), ma non includeva un dominio specificamente dedicato ai docenti. Qualsiasi estensione in tal senso avrebbe rischiato di forzare il modello esistente.
- **Limitato riuso dei componenti**: schemi, parametri e risposte erano presenti, ma non organizzati in modo centralizzato. Alcuni concetti ricorrenti (paginazione, identificativi, errori comuni) venivano definiti più volte o con leggere varianti, aumentando il rischio di incoerenze.
- **Processo manuale**: l'artefatto principale `openapi.yaml` veniva modificato direttamente, senza un processo di generazione automatica. Ogni modifica strutturale era potenzialmente rischiosa e difficile da verificare, perché non esistevano pipeline dedicate al bundling e alla validazione.

Queste considerazioni hanno messo in luce la necessità di un intervento più strutturale, volto non soltanto ad aggiungere nuove rotte, ma a ripensare l'organizzazione complessiva della specifica per renderla estendibile verso il dominio docente.

4.1.1 Aggiornamento dello standard OpenAPI

Il primo intervento ha riguardato l'aggiornamento della specifica alla versione OpenAPI 3.1.1, l'ultima versione stabile disponibile al momento [2].

L'adozione della versione 3.1.1 ha rappresentato un passaggio fondamentale per progettare delle API in modo più moderno e robusto. Questa versione introduce un allineamento completo agli standard più recenti di *JSON Schema*, consentendo di descrivere i modelli dati in maniera più chiara, precisa e riusabile rispetto alla precedente 3.0.3. L'eliminazione di diverse limitazioni, come parole chiave proprietarie o workaround necessari per gestire casi particolari, ha reso possibile definire strutture dati uniformi nell'intero progetto.

Il passaggio alla versione 3.1.1 ha inoltre incrementato la coerenza del contratto API, grazie a una validazione più espressiva. Tra gli elementi che hanno contribuito a semplificare la definizione delle nuove specifiche troviamo una gestione più ordinata dei riferimenti e la possibilità di inserire esempi direttamente all'interno dei modelli.

Nel complesso, l'aggiornamento ha fornito un ambiente più solido per la progettazione e una maggiore compatibilità con gli strumenti moderni di validazione, generazione del codice e produzione della documentazione.

Questo ha migliorato in modo significativo la qualità complessiva della specifica e ha facilitato la definizione delle nuove API dedicate a *PoliTO Faculty*.

Dal punto di vista del processo, l'aggiornamento è frutto di una scelta progettuale consapevole e non di un semplice adeguamento tecnologico. In fase iniziale è stata infatti valutata anche l'ipotesi di mantenere la specifica esistente in versione 3.0.3,

limitandosi ad aggiungere le nuove operazioni necessarie a *PoliTO Faculty*. Questa opzione avrebbe semplificato la gestione della retrocompatibilità, riducendo il rischio di introdurre problemi sulle API già utilizzate dall'applicazione *PoliTO Students*.

La discussione si è quindi concentrata sul bilanciamento tra la stabilità nel breve periodo e la qualità complessiva del sistema nel medio termine. Mantenere la specifica in versione 3.0.3 avrebbe garantito continuità con lo stato attuale, ma avrebbe complicato l'introduzione di meccanismi di validazione automatica e l'adozione degli strumenti più recenti per la generazione del codice e della documentazione. L'aggiornamento alla versione 3.1.1 è stato pertanto preferito, in quanto in grado di offrire un'esperienza di sviluppo più solida e un controllo più rigoroso sulla coerenza della specifica, aspetti considerati fondamentali per assicurare la manutenibilità della specifica nel tempo.

4.1.2 Passaggio da specifica monolitica a struttura modulare

In parallelo all'aggiornamento dello standard, è stata ripensata l'organizzazione dell'artefatto principale, passando da una specifica monolitica a una struttura modulare, pensata per rendere più semplice e sicuro il lavoro collaborativo sul progetto.

Il file `openapi.yaml` non viene più modificato direttamente, ma è diventato un artefatto generato a partire da una struttura modulare

A tal fine, il file `openapi.yaml`, che inizialmente definiva l'intera specifica, è stato riorganizzato in tre sezioni principali, corrispondenti con i domini funzionali delle API incluse in ciascuna area:

1. **Students**: raccoglie le API esistenti relative al dominio studenti, utilizzate esclusivamente dall'applicazione *PoliTO Students*.

2. **Faculty**: comprende le nuove API sviluppate per supportare i servizi specifici dell'applicazione *PoliTO Faculty*, utilizzate esclusivamente da quest'ultima..

3. **Common**: contiene tutte le API condivise dalle due applicazioni.

La suddivisione della specifica nei domini *Common*, *Students* e *Faculty* è stata definita in accordo con il dipartimento ISIAD, con l'obiettivo di riflettere in modo chiaro la distinzione tra funzionalità condivise e funzionalità specifiche delle due applicazioni *mobile*. Durante la fase di progettazione è stata presa in considerazione anche un'organizzazione alternativa, basata su una separazione preliminare per funzionalità (ad esempio "esami", "aule", "notifiche") e, solo in un secondo momento, per tipologia di utenza (esami studenti, esami docenti, e così via).

Questo approccio è stato però scartato per diverse ragioni. In primo luogo, non tutti i casi d'uso presentano una separazione netta tra ambito studenti e ambito docenti: alcune risorse, come emergenze, mappe o news, sono per loro natura trasversali e, in un'organizzazione basata sulle funzionalità, sarebbero state esposte al rischio di duplicazioni o di una frammentazione tra più file. In secondo luogo, una struttura centrata sulle funzionalità avrebbe reso meno immediato individuare il perimetro effettivo di ciascuna applicazione, complicando la collaborazione tra i team che lavorano su *Students* e su *Faculty*. Infine, la presenza di un dominio esplicito per i componenti comuni riduce il rischio di definire varianti leggermente diverse della stessa entità in punti diversi della specifica, favorendo invece il riuso controllato di schemi, parametri e risposte. La scelta di organizzare la specifica per domini *Common*, *Students* e *Faculty* rappresenta quindi un compromesso tra chiarezza architetturale, manutenibilità e possibilità di evoluzione nel lungo periodo.

4.2 Modellazione delle entità e flussi di comunicazione

Il lavoro di riorganizzazione del progetto, ha permesso di modellare le entità esistenti e renderle compatibili con le nuove. Questo ha portato alla definizione di tre insiemi di schemi, organizzati per dominio (common, faculty, students).

4.2.1 Entità comuni

Nel gruppo di entità comuni sono stati definiti i modelli che rappresentano concetti condivisi, tra cui:

- **Booking** e **PlaceBooking**, che descrivono le prenotazioni di aule e spazi, con i relativi metadati (intervallo temporale, risorsa, stato);
- **Course**, **CourseDirectoryContent**, **Exam**, che rappresentano corsi, materiali didattici e appelli d'esame;
- **NewsItem** e **Person**, utilizzati per la pubblicazione di comunicazioni e la rappresentazione anagrafica degli utenti;

Gli schemi comuni sono stati modellati esplicitando i campi obbligatori (`required`) e sfruttando costrutti come `oneOf` e `nullable` per gestire varianti e campi opzionali all'interno delle stesse strutture dati.

4.2.2 Entità del dominio studenti

I componenti relativi al dominio studenti sono stati oggetto di un intervento limitato, finalizzato esclusivamente a garantirne la piena compatibilità con la nuova versione dello standard OpenAPI. Le modifiche introdotte sono state minime e hanno interessato quasi esclusivamente la gestione dei campi nullabili, senza incidere sulla logica complessiva o sul comportamento funzionale dei componenti.

4.2.3 Entità del dominio docenti

L'ultimo gruppo introduce le entità necessarie a coprire le funzionalità destinate ai docenti, assenti nella specifica originale. I modelli principali includono:

- **FacultyProfile**, che descrive il profilo docente (informazioni anagrafiche, struttura di afferenza, canali di contatto);
- **FacultyExam**, che rappresenta gli appelli lato docente, con informazioni su corso, data, iscritti e stato;
- **FacultyCourseStudent** e **Collaborator**, utilizzati per modellare la composizione di un incarico didattico (studenti e collaboratori associati);
- **CourseLecture**, che estende il concetto di lezione lato docente, collegandolo al contesto del corso e alle responsabilità didattiche;
- **SignableDocument** e gli altri schemi che descrivono il flusso di firma digitale (richiesta di firma, stato della sessione, esito dell'operazione).

Questi modelli sono stati progettati per supportare i flussi introdotti dal nuovo dominio *faculty*: gestione degli incarichi, comunicazioni agli studenti dei corsi, pianificazione delle attività in calendario e firma digitale dei documenti.

4.2.4 Parametri, risposte e flussi di comunicazione

Accanto agli schemi, la specifica definisce blocchi dedicati a parametri e risposte riusabili:

- `parameters.yaml` centralizza gli identificativi (*id* di corsi, esami, documenti), i parametri di paginazione (`page`, `pageSize`) e i filtri di ricerca (ad esempio per sito, edificio, categoria di luogo);
- `responses.yaml` raccoglie le risposte standard per gli scenari di successo e di errore, incluse quelle specializzate per flussi particolari (MFA e firma digitale).

A partire da questo modello dati, sono stati tracciati i principali flussi di comunicazione tra backend e applicazione mobile, tra cui:

- la gestione della carriera e degli esami (consultazione della lista, prenotazione e spostamento di un appello);
- la pianificazione e prenotazione di spazi (ricerca disponibilità, creazione e cancellazione di una prenotazione);
- la gestione degli incarichi didattici lato docente (consultazione degli studenti iscritti, gestione dei collaboratori, invio di comunicazioni);
- il ciclo completo di firma digitale (lista documenti, avvio della sessione, verifica dell'OTP, firma con PIN);
- la gestione di emergenze e guasti (consultazione delle comunicazioni di emergenza, apertura e tracciamento di segnalazioni).

In questo modo la modellazione delle entità non rimane astratta, ma è direttamente collegata ai casi d'uso che l'applicazione deve supportare.

4.3 Progettazione delle API RESTful

Una volta definito il modello dati, il passo successivo ha riguardato la progettazione dei singoli endpoint, seguendo convenzioni REST e sfruttando la nuova struttura modulare della specifica [11].

4.3.1 Pattern REST adottati

La progettazione delle interfacce segue alcune convenzioni REST condivise, finalizzate a garantire predicibilità e facilità di consumo [12]:

- **Risorse e sottorisorse**: i path esprimono esplicitamente la relazione tra risorse, ad esempio:
  - `/courses/{courseId}/lectures`
  - `/faculty/courses/{courseId}/students`
- **Verbi HTTP con semantica chiara**: `GET` è utilizzato per la lettura, `POST` per la creazione o l'attivazione di azioni non idempotenti (ad esempio `/faculty/exams/{examId}/notify`), `PUT` per operazioni idempotenti di upsert, `DELETE` per la rimozione di risorse.

- **Filtri e paginazione**: i parametri di query sono utilizzati per filtrare gli elenchi e per gestire la paginazione.

Nel dominio *Faculty* sono state progettate le nuove API per:

- la gestione del profilo docente e delle pubblicazioni:
  - `/faculty/me`
  - `/faculty/me/publications`
- la gestione degli incarichi didattici:
  - `/faculty/courses/{courseId}/students`
  - `/faculty/courses/{courseId}/collaborators`
  - `/faculty/courses/{courseId}/message`
- la gestione degli appelli:
  - `/faculty/exams`
  - `/faculty/exams/{examId}`
  - `/faculty/exams/{examId}/notify`
- il ciclo di firma digitale:
  - `/faculty/signing/documents`
  - `/faculty/signing/sessions/{sessionId}/verify-otp`
  - `/faculty/signing/sessions/{sessionId}/sign`
- i calendari e gli eventi:
  - `/faculty/calendars`
  - `/faculty/calendars/{calendarId}/events`

La progettazione di questi endpoint è stata guidata dai flussi individuati nella fase di modellazione: a ogni caso d'uso è stato associato un insieme di operazioni, mantenendo coerenza con gli endpoint già esistenti per gli studenti quando possibile.

L'introduzione delle nuove API per il dominio Faculty è stata l'occasione per rendere più esplicite alcune linee guida di progettazione già presenti, in modo talvolta implicito, nella specifica esistente. L'obiettivo principale non era introdurre pattern radicalmente diversi, ma consolidare uno stile uniforme che rendesse la lista di endpoint più prevedibile e semplice da utilizzare nel tempo.

In assenza di convenzioni condivise, la crescita progressiva di una specifica rischia infatti di produrre interfacce eterogenee: path strutturati in modo diverso per

rappresentare concetti simili, utilizzo non coerente dei verbi HTTP, duplicazioni di logica tra risorse che appartengono a domini correlati. Per ridurre questo rischio, la progettazione delle operazioni dedicate a *PoliTO Faculty* è stata guidata dal principio di mantenere, quando possibile, lo stesso modello concettuale già adottato per gli studenti, estendendolo solo nei casi in cui le esigenze del personale docente richiedessero funzionalità aggiuntive.

In pratica, ciò si traduce in alcune scelte ricorrenti: modellare i principali elementi del dominio come risorse accessibili tramite URL leggibili, utilizzare le sottorisorse per rappresentare relazioni gerarchiche (ad esempio tra corsi, appelli ed elenchi di studenti) e mantenere una semantica dei verbi HTTP coerente tra i diversi domini applicativi. Questo approccio facilita il lavoro dei team che sviluppano e mantengono le applicazioni mobili, perché riduce la necessità di "imparare" convenzioni diverse per studenti e docenti e rende più agevole l'estensione futura dell'API senza introdurre rotture di coerenza.

4.4 Sicurezza e autenticazione

La progettazione delle API ha dovuto rispettare i vincoli di sicurezza definiti a livello di requisiti non funzionali, garantendo protezione dei dati, gestione corretta delle sessioni e tracciabilità delle operazioni sensibili.

4.4.1 Schema di sicurezza centralizzato

Per quanto riguarda la sicurezza degli endpoint, è stato definito uno schema di autenticazione `bearerAuth`, referenziato dalle singole operazioni che richiedono accesso autenticato [13]. In questo modo:

- la modalità di autenticazione è descritta in un unico punto, riducendo il rischio di incoerenze;
- eventuali variazioni future (ad esempio l'introduzione di nuovi schemi) potranno essere gestite senza modificare tutti i path.

La quasi totalità delle API dei domini *Students* e *Faculty* utilizza questo schema, mentre alcune operazione di autenticazione (come il login) restano accessibili in modalità anonima.

La scelta di adottare queste convenzioni non è stata scontata, ma nasce dal confronto con alcuni pattern alternativi emersi durante la revisione della specifica esistente. Una prima possibilità consisteva nel mantenere endpoint dal naming più "procedurale", in cui l'azione prevale sul concetto di risorsa (ad esempio /getExamList, /doBooking, /sendMessageToStudents). Questo tipo di interfaccia

risulta inizialmente intuitivo, ma tende a generare cataloghi di API poco omogenei, difficili da estendere e meno comprensibili per chi non conosce la storia evolutiva del sistema.

Un secondo rischio riguardava la coesistenza di stili diversi tra il dominio studenti e il nuovo dominio docenti, con path simili modellati in modo non uniforme. Per evitare questa situazione, la progettazione delle nuove rotte Faculty è stata guidata dal principio di riutilizzare, quando possibile, gli stessi concetti e la stessa struttura delle API già esistenti per gli studenti, estendendole solo dove strettamente necessario. In questo modo si è ottenuto un catalogo di endpoint più prevedibile, in cui la relazione tra risorse e sottorisorse è leggibile direttamente dall'URL e la semantica dei verbi HTTP rimane coerente tra i diversi domini applicativi. Questa coerenza semplifica il lavoro sia dei team che mantengono le API, sia degli sviluppatori che devono integrarle all'interno delle applicazioni mobili.

4.5 Struttura del repository

L'ultimo passaggio del processo di progettazione riguarda la finalizzazione della struttura del repository della specifica. L'obiettivo non era solo descrivere le API, ma renderne sostenibile l'evoluzione nel tempo.

4.5.1 Struttura dei sorgenti

La cartella `src/` contiene tutti i file sorgente della specifica:

- `src/index.yaml` come principale punto di accesso, con i metadati globali, la lista dei path e dei componenti e la definizione dei gruppi di tag;
- i path organizzati per dominio funzionale:
  - `src/paths/common/`
  - `src/paths/students/`
  - `src/paths/faculty/`
- i modelli di dominio:
  - `src/components/schemas/common.yaml`
  - `src/components/schemas/students.yaml`
  - `src/components/schemas/faculty.yaml`
- gli elementi riusabili trasversali:
  - `src/components/parameters.yaml`

- `src/components/responses.yaml`
- `src/components/security-schemes.yaml`

All'interno di `src/index.yaml` è stata inoltre riorganizzata la struttura dei tag utilizzati nella documentazione OpenAPI. I tag globali riprendono la stessa suddivisione in domini funzionali adottata per i sorgenti (*Common*, *Students*, *Faculty*) e vengono assegnati alle operazioni in base al contesto applicativo.

L'adozione di questa organizzazione ha reso possibile l'aggiornamento della specifica OpenAPI, migliorandone la manutenibilità e assicurando una maggiore scalabilità nel lungo periodo.

Le Figure 4.1 e 4.2 mostrano in modo sintetico la struttura della repository prima e dopo l'introduzione di tali modifiche.

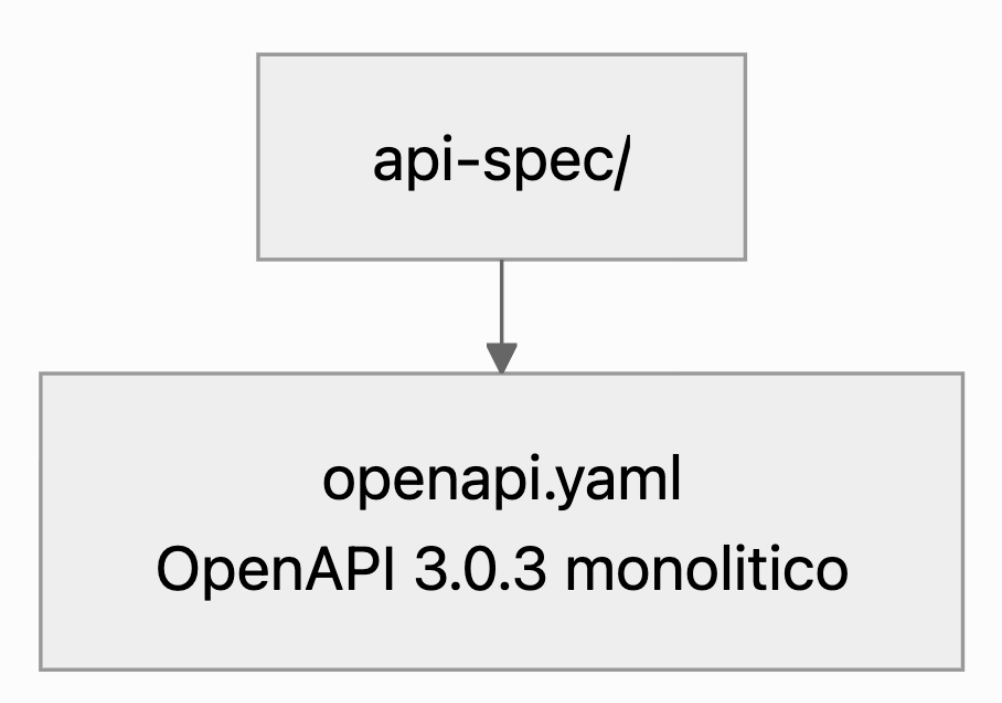

Figura 4.1: Specifica monolitica

4.5.2 Tag e organizzazione della documentazione

Nel contesto di OpenAPI, i tag sono etichette assegnate alle operazioni con lo scopo di raggrupparle in insiemi coerenti e rendere più semplice la consultazione della documentazione. Una buona struttura di tag aiuta a orientarsi tra le API, separando i diversi ambiti funzionali e riducendo la complessità percepita da chi deve integrare il sistema.

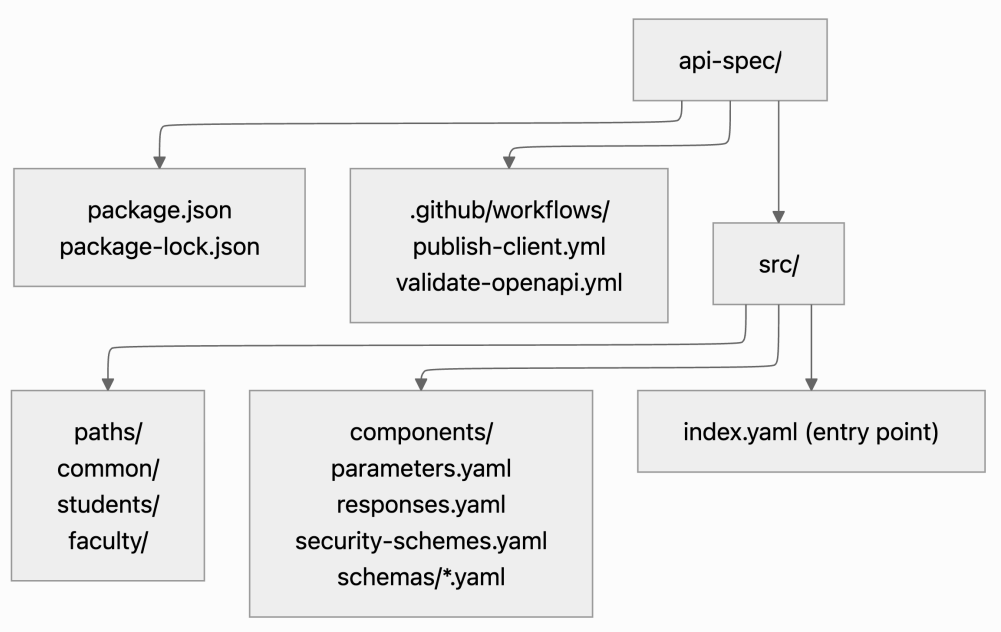

Figura 4.2: Struttura modulare

Nel progetto `api-spec` i tag definiti in `src/index.yaml` riprendono la suddivisione in domini funzionali adottata per i sorgenti (*Common*, *Students*, *Faculty*) e vengono assegnati alle operazioni in base al contesto applicativo. L'estensione `x-tagGroups` permette inoltre di raggruppare i tag in sezioni logiche, così che gli strumenti di visualizzazione della specifica presentino blocchi distinti per le API comuni, per quelle rivolte agli studenti e per quelle dedicate ai docenti, ciascuno con il proprio insieme coerente di path e modelli.

4.5.3 Tooling e generazione automatica

Nel file `package.json` sono stati definiti gli script necessari per:

- installare le dipendenze (`npm ci`);
- generare e validare il bundle (`npm run bundle`, `npm run validate`, `npm run bundle:verify`).

4.5.4 Pipeline di validazione e qualità del client OpenAPI

La qualità della specifica è supportata da *workflow* GitHub dedicati. In particolare:

- un workflow di pubblicazione del client, che esegue il bundling e la validazione della specifica prima di generare e pubblicare il client TypeScript;
- un workflow di validazione (`validate-openapi`) eseguito sulle pull request, che lancia il bundling tramite `swagger-cli` e la validazione tramite `openapi-generator-cli`, bloccando eventuali regressioni strutturali [14].

In questo modo il processo di progettazione delle API è integrato con la pipeline di integrazione continua: ogni modifica alla specifica deve superare la validazione prima di essere aggiunta al *branch* principale.

Capitolo 5

Implementazione e Integrazione

In questo capitolo viene descritto come la specifica OpenAPI, progettata nel capitolo precedente, è stata portata all'interno dell'applicazione *PoliTO Faculty*. L'obiettivo è illustrare il percorso che va dall'artefatto di specifica alla disponibilità di un client TypeScript riutilizzabile, fino alla sua integrazione in un prototipo già esistente basato su dati mock.

5.1 Ambiente di sviluppo

Le attività di implementazione si sono svolte in due repository separati:

- il repository della specifica (*api-spec*), in cui è mantenuta la definizione OpenAPI modulare e in cui vengono eseguite le operazioni di bundling, validazione e generazione del client;
- il repository dell'applicazione *PoliTO Faculty* (*faculty-app*), contenente il prototipo mobile con le schermate e i flussi utente già modellati tramite dati mock.

5.1.1 Tooling per la generazione del client

Nel progetto *api-spec* sono stati introdotti gli strumenti necessari per rendere ripetibile il processo di generazione del client:

- utilizzo di `Node 18` come versione di riferimento per gli script di progetto [15];
- definizione di un `package.json` con gli script necessari per eseguire il bundling, la validazione della specifica e la generazione del client;

- dipendenze verso `swagger-cli` e `openapi-generator-cli`, necessarie per eseguire gli script [14];
- configurazione di `.gitignore` per mantenere il repository pulito ed escludere i file generati.

In questo modo, l'applicazione non accede direttamente ai file *YAML* della specifica, ma lavora su un artefatto stabile: il client TypeScript generato a partire dalla versione corrente della PoliTO API.

5.1.2 Ambiente di sviluppo e testing locale

Durante la fase di implementazione è stato utilizzato l'IDE *Visual Studio Code*, affiancato da emulatori Android e iOS per la verifica funzionale dei flussi dopo ogni modifica al codice [16].

Per il testing locale sono stati utilizzati dati di esempio forniti da un server avviato tramite *Prism CLI*, che, a partire dalla specifica OpenAPI del progetto, genera un server mock in grado di esporre gli endpoint presenti nella specifica e restituire risposte con la struttura definita [17]. Questa configurazione ha reso possibile verificare in modo incrementale i singoli flussi implementati, senza dipendere da ambienti di test o da dati reali.

Per rendere più rapido il processo di validazione, l'applicazione è stata eseguita in modalità *dev*, sfruttando la funzione di *Fast Refresh* di React Native [6]. Tale funzionalità permette di visualizzare quasi in tempo reale le modifiche apportate al codice TypeScript, senza la necessità di ricompilare o riavviare completamente l'app. Il mantenimento dello stato corrente dell'interfaccia ha ulteriormente ridotto i tempi di attesa dopo ogni modifica, contribuendo a velocizzare in modo significativo l'intero ciclo di sviluppo.

5.1.3 Scelte architetturali per l'integrazione delle API

L'integrazione delle nuove API nel prototipo *PoliTO Faculty* non si è limitata all'invocazione dei singoli endpoint, ma ha richiesto alcune scelte architetturali per garantire coerenza con l'applicazione esistente e una buona manutenibilità del codice nel medio periodo. In questa sezione vengono descritte le principali decisioni progettuali adottate lato client e le motivazioni che ne hanno guidato l'introduzione.

Utilizzo di React Query e hook

Per la gestione delle chiamate di rete è stato confermato l'utilizzo di *React Query* in combinazione con hook dedicati, seguendo lo stesso approccio già consolidato

all'interno dell'applicazione PoliTO Students. Questa scelta è stata dettata da ragioni principalmente pragmatiche.

Da un lato, *React Query* si era già dimostrato uno strumento stabile e privo di criticità nel contesto dell'app esistente, fornendo astrazioni utili per gestire gli stati di caricamento, errore e aggiornamento dei dati remoti. Dall'altro, mantenere lo stesso stack tecnologico ha permesso di ridurre la curva di apprendimento per gli sviluppatori che in futuro dovranno lavorare su entrambe le applicazioni, evitando di introdurre pattern alternativi (come librerie di stato globale o soluzioni personalizzate per il *data fetching*) che avrebbero reso il codice più eterogeneo.

Gli hook specifici per ciascun caso d'uso incapsulano la logica di chiamata alle API e di gestione dello stato, esponendo ai componenti di interfaccia un insieme limitato di proprietà già pronte all'uso (ad esempio dati normalizzati, flag di caricamento e funzioni di aggiornamento). In questo modo i componenti rimangono relativamente semplici e focalizzati sulla presentazione, mentre la logica di integrazione è centralizzata e più facilmente testabile.

Client TypeScript generato dalla specifica

Un secondo elemento centrale dell'architettura lato client è l'utilizzo di un client TypeScript generato automaticamente a partire dalla specifica OpenAPI. Invece di definire manualmente le chiamate HTTP per ciascun endpoint, il progetto sfrutta uno strumento di generazione in grado di produrre funzioni tipizzate e modelli di dato coerenti con quanto definito nel file di specifica.

Questa scelta è in linea con l'approccio *API-first* adottato nel progetto: la specifica costituisce la fonte di verità principale e sia il backend sia il frontend derivano la propria struttura da essa. Dal punto di vista pratico, la generazione del client riduce il rischio di divergenze tra contratto e implementazione (ad esempio campi mancanti o rinominati) e permette di individuare più rapidamente eventuali regressioni, grazie al supporto del type checker di TypeScript. Inoltre, l'evoluzione futura delle API risulta semplificata: una modifica alla specifica si traduce in una nuova generazione del client, limitando gli interventi manuali al codice di integrazione e ai componenti che utilizzano i relativi modelli.

Layer di adattamento tra modelli API e modelli di interfaccia

Per evitare un accoppiamento eccessivo tra la struttura delle API e i componenti di interfaccia, è stato introdotto un layer di adattamento dedicato alla trasformazione dei dati ricevuti dal server in modelli più adatti alla logica di presentazione. In pratica, le strutture generate a partire dalla specifica vengono convertite in oggetti di dominio semplificati, utilizzati all'interno dell'applicazione mobile.

Questo approccio consente di isolare i componenti UI dai dettagli di rappresentazione scelti a livello di API (ad esempio formati di date, codici enumerativi

o campi opzionali legati a esigenze backend), mantenendo nello stesso tempo un punto unico in cui applicare eventuali regole di normalizzazione o arricchimento dei dati. In prospettiva, la presenza di questo layer rende meno impattante l'evoluzione della specifica: finché il contratto resta compatibile a livello concettuale, buona parte delle modifiche può essere assorbita all'interno degli adapter, senza dover propagare sistematicamente le differenze a tutti i componenti dell'interfaccia.

Utilizzo di un server mock basato su Prism

Infine, l'integrazione delle API è stata supportata dall'utilizzo di un server mock generato a partire dalla specifica tramite *Prism*. In fase di tesi non era disponibile il tempo necessario per implementare e pubblicare un backend reale che esponesse gli endpoint definiti nello standard, ma era comunque fondamentale poter verificare i flussi applicativi end-to-end dal punto di vista del client.

L'adozione di Prism ha permesso di simulare il comportamento delle API direttamente sulla base della specifica OpenAPI, garantendo coerenza tra il contratto documentato e le risposte restituite all'applicazione. In questo modo è stato possibile iterare rapidamente sull'integrazione: una modifica alla specifica comportava la rigenerazione del client e l'aggiornamento del mock, consentendo di testare immediatamente l'effetto sui flussi dell'applicazione attraverso gli emulatori. Questa modalità di lavoro è risultata particolarmente efficace per un prototipo, perché ha permesso di concentrarsi sulla qualità del contratto e sull'esperienza d'uso lato mobile, rimandando a una fase successiva l'implementazione definitiva del backend.

5.2 Implementazione dei moduli principali

Il punto di partenza lato frontend era un prototipo dell'applicazione *PoliTO Faculty* in cui i dati relativi al profilo docente, agli incarichi didattici, alla firma digitale e alle prenotazioni venivano forniti da un unico contesto globale. Questo provider esponeva al resto dell'applicazione strutture dati mock, definite localmente e scollegate da qualsiasi contratto API. Non erano presenti né servizi intermedi né hook dedicati: i componenti di interfaccia leggevano direttamente dal contesto, mescolando logica di presentazione e logica di accesso ai dati.

L'integrazione con il client TypeScript generato dalla specifica OpenAPI ha richiesto un ripensamento dell'architettura di accesso ai dati. Si è adottato uno schema a tre livelli:

$$\texttt{Componenti UI} \rightarrow \texttt{React Query Hooks} \rightarrow \texttt{Client API generato}$$

I componenti non interagiscono più con provider globali o chiamate HTTP dirette, ma utilizzano hook sviluppati per gestire la comunicazione con il server. Questi hook

incapsulano la logica di interrogazione e aggiornamento dei dati, appoggiandosi al client TypeScript fornito dalla libreria `@polito/api-client` [18]. In questo modo è stato possibile:

- eliminare la dipendenza dal contesto globale contenente i dati mock;
- separare in modo netto la logica di presentazione dalla logica di integrazione con le API;
- sfruttare la type safety ottenuta a partire dalla specifica OpenAPI.

La configurazione globale di React Query, definita in un `QueryClient` condiviso, gestisce in modo uniforme caching, politiche di *retry*, invalidazione e gestione degli errori [19]. Ogni hook definisce una `queryKey` costante, uno `queryFn` che invoca il client generato e, dove necessario, una pipeline di trasformazioni.

Nei paragrafi seguenti vengono descritti i moduli principali coinvolti dall'integrazione.

Profilo docente

Nel prototipo iniziale, le informazioni di profilo del docente (dati anagrafici, struttura di afferenza, contatti) venivano esposte al resto dell'applicazione tramite oggetti mock costruiti ad hoc per le schermate.

Con l'introduzione del client generato, il modulo è stato ristrutturato come segue:

- è stato introdotto un hook dedicato alla lettura del profilo che istanzia il client `FacultyProfileApi` e incapsula la chiamata all'endpoint corrispondente, restituendo al componente i soli dati necessari, insieme agli stati `isLoading`, `isError` e `refetch`;
- la risposta del client viene elaborata in una pipeline che estrae il campo `data`, esegue eventuali trasformazioni per la formattazione (es. normalizzazione di stringhe, composizione di campi derivati) e infine mappa le informazioni in modo da utilizzare lo stesso formato del modello già utilizzato dal componente del profilo.

La schermata di profilo, mostrata in Figura 5.1, utilizza l'hook per popolare i dati anagrafici e di contatto del docente. La vista rimane invariata rispetto al prototipo iniziale, ma i valori visualizzati provengono ora dalle entità esposte dalle API del dominio *Faculty*.

La schermata di profilo continua così ad avere la stessa logica, ma i valori provengono ora dal contratto definito in OpenAPI e non più da oggetti mock definiti manualmente.

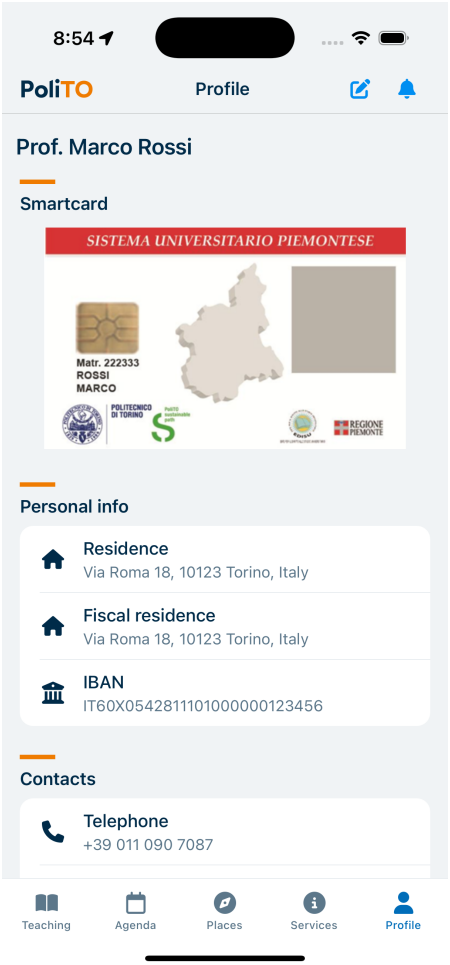

Figura 5.1: Schermata principale del profilo docente nell'applicazione *PoliTO Faculty*.

Incarichi didattici e gestione corsi

Anche gli incarichi didattici e, più in generale, il dominio *corsi* facevano affidamento sul contesto, che esponeva liste di corsi, lezioni e studenti iscritti tramite oggetti mock.

La nuova integrazione ha introdotto due livelli di accesso ai dati:

- un set di hook responsabili rispettivamente per le operazioni di lettura sui corsi e sui relativi studenti;
- l'utilizzo combinato dei client generati `CommonCoursesApi` e di un wrapper `FacultyCoursesApi`, che incapsula le operazioni specifiche del dominio docente

sugli stessi corsi (ad esempio azioni che riguardano solo i docenti, come la gestione dei collaboratori).

La schermata *Teaching*, riportata in Figura 5.2, rappresenta uno degli esiti più rilevanti dell'integrazione con la nuova specifica. Le liste degli insegnamenti e degli appelli d'esame sono infatti alimentate dagli hook di React Query che interrogano i client `CommonCoursesApi` e `FacultyExamsApi`, garantendo il rispetto del contratto OpenAPI e mostrando lo stesso tipo di informazioni.

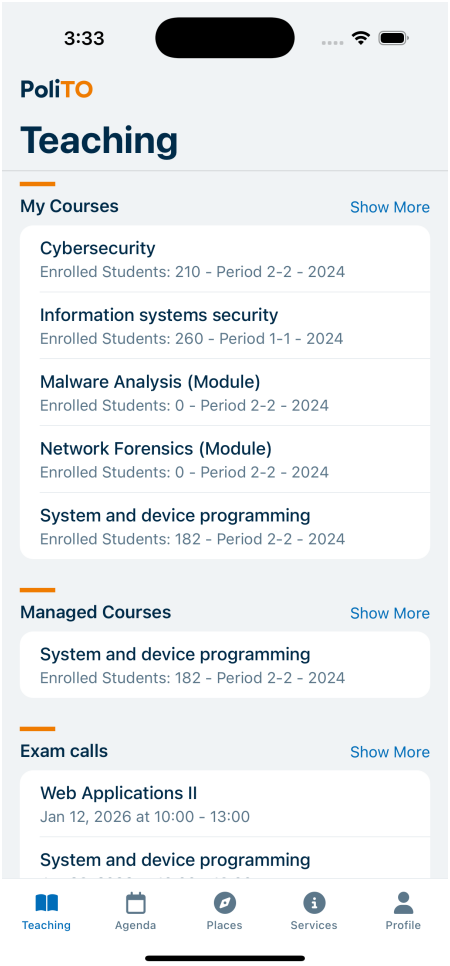

Figura 5.2: Schermata *Teaching* con i corsi e gli appelli d'esame del docente.

Gli hook adottano il seguente pattern: istanziano il client, eseguono la chiamata API nel `queryFn` di React Query e applicano una serie di trasformazioni sui dati. Queste trasformazioni sono state isolate in funzioni di conversione dedicate, che si occupano di:

- mappare le enumerazioni e i codici restituiti dall'API in etichette testuali comprensibili e adatte all'interfaccia utente;
- convertire le date ISO in oggetti `Date` e in stringhe formattate per la visualizzazione;
- appiattire strutture annidate (ad esempio combinando informazioni su sede, edificio e aula in un'unica stringa descrittiva);
- gestire `null` e campi opzionali assegnando valori di default adeguati.

Le schermate degli incarichi (lista degli insegnamenti, dettaglio del corso, elenco degli studenti e dei collaboratori) sono state progressivamente migrate a questo modello: al posto di leggere direttamente dai dati mock, consumano gli hook React Query, beneficiando della cache, del recupero in caso di errore e del supporto alla funzionalità di *pull-to-refresh*.

Firma digitale

La firma digitale rappresenta uno dei flussi più delicati per i docenti, perché coinvolge documenti amministrativi e richiede un processo a più step (inizializzazione, verifica tramite OTP, conferma tramite PIN). Nel prototipo iniziale, anche questo flusso era simulato tramite dati mock e transizioni di stato gestite interamente lato client.

L'integrazione con le API del dominio *Faculty* ha portato alla definizione di hook dedicati al ciclo di firma, che incapsulano le chiamate verso gli endpoint `/faculty/signing/*`. A livello concettuale, ogni passaggio è modellato da un hook che:

- **Accesso al servizio:** recupera il client per la firma digitale utilizzando una *factory function* dedicata;
- **Gestione operazioni:** esegue le diverse fasi del processo (inizializzazione sessione, verifica OTP, firma) tramite una `mutation` di React Query;
- **Mapping degli errori:** intercetta le eccezioni API (es. OTP non valido, sessione scaduta) e le converte, tramite funzioni ausiliarie, in messaggi chiari e stati gestibili dall'interfaccia utente.

La vista principale del modulo di firma digitale, riportata in Figura 5.3, distingue i documenti ancora da firmare da quelli già firmati. Il recupero dei dati avviene tramite una query all'endpoint del dominio *Faculty*, filtrando poi i risultati in base al loro stato.

Una volta selezionato un documento, il flusso di firma, prevede: invio dell'OTP, inserimento del codice ricevuto e conferma tramite PIN. La Figura 5.4 mostra la

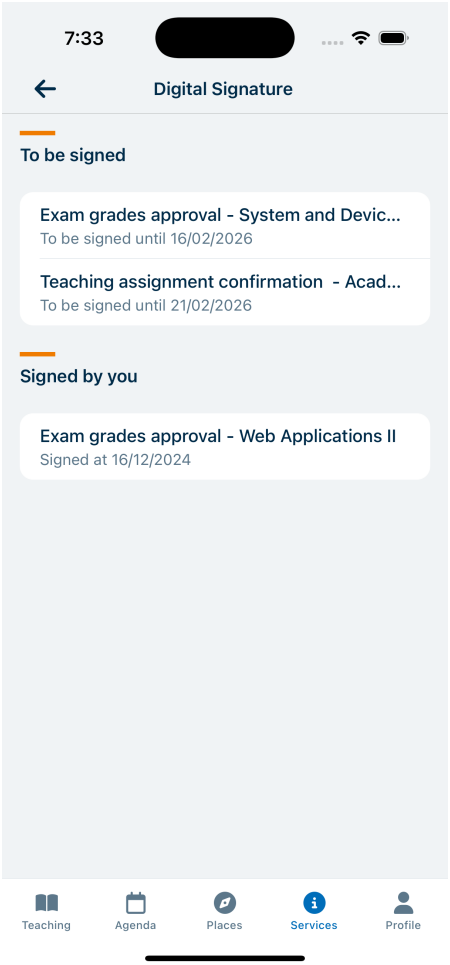

Figura 5.3: Lista dei documenti da firmare e dei documenti già firmati nel modulo di firma digitale.

schermata di inserimento dell'OTP, che riflette lo stato restituito dalla risposta API (invio del codice al recapito impostato) e permette di gestire esplicitamente errori e tentativi falliti.

La Figura 5.5 illustra l'ultimo passaggio del processo, ossia l'inserimento del PIN, la cui validazione conclude il flusso previsto per la firma del documento.

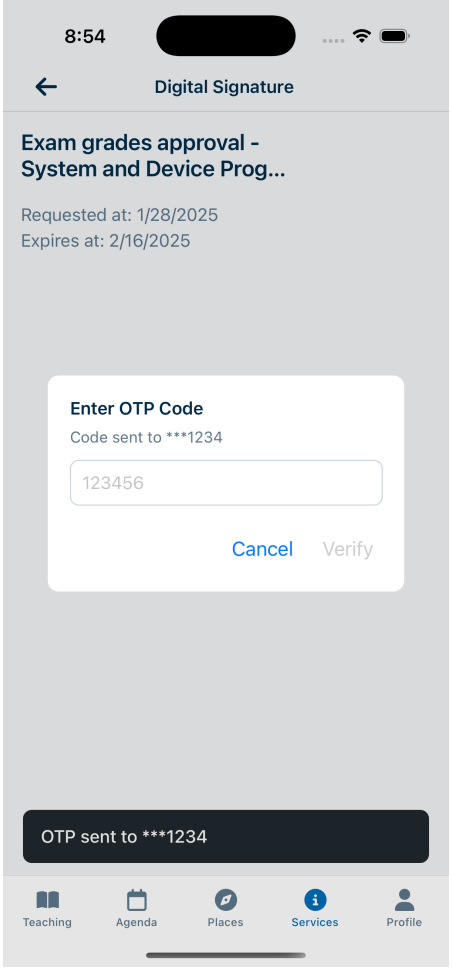

Figura 5.4: Step di inserimento dell'OTP nel flusso di firma digitale.

La UI non interagisce direttamente con il client, ma si appoggia esclusivamente agli hook, limitandosi a gestire i diversi stati dell'operazione (*idle*, caricamento, successo ed errore). Questo consente di mantenere centralizzato il flusso di firma definito nella specifica OpenAPI, evitando che venga inclusa della logica all'interno dei componenti.

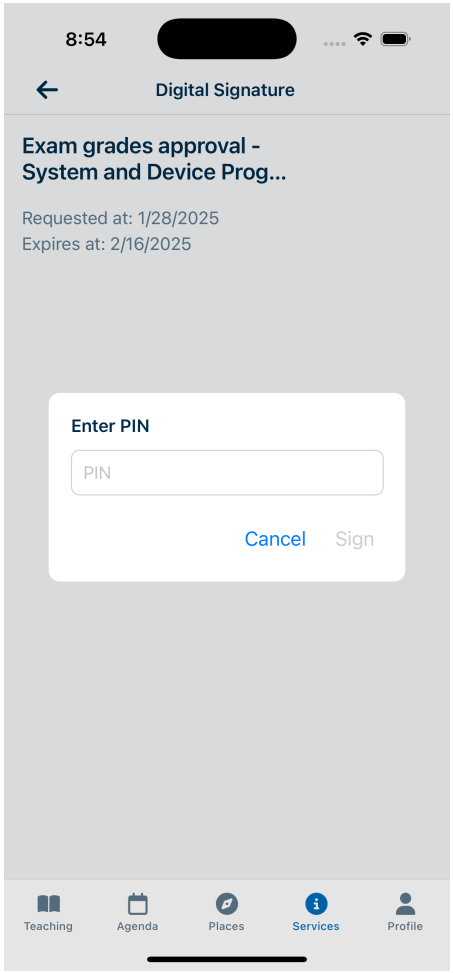

Figura 5.5: Step di inserimento del PIN nel flusso di firma digitale.

Prenotazioni di spazi e aule

Infine, il modulo delle prenotazioni di spazi è passato da una gestione completamente mock ad un'integrazione con le API. In origine le richieste visualizzate nelle schermate erano oggetti statici forniti dall'apposito contesto, utili per validare i flussi di interazione ma privi di qualsiasi legame con i dati reali.

La nuova implementazione si basa su un insieme di hook che coprono i principali casi d'uso (lista delle prenotazioni del docente, creazione, aggiornamento e cancellazione di una prenotazione) e utilizzano il client generato per le prenotazioni. Le `mutation` che eseguono operazioni di creazione o annullamento invalidano automaticamente le query corrispondenti, così che la lista visualizzata rimanga aggiornata rispetto allo stato restituito dal backend.

Il dettaglio di una singola prenotazione, riportato in Figura 5.6, mostra il risultato di questa integrazione: tutti i campi visualizzati (intervallo temporale, luogo, capacità, attrezzature e servizi) derivano dal modello restituito dal client e trasformato dal `placeBookingAdapter`. L'azione di cancellazione è collegata a un endpoint dedicato e gestita tramite una `mutation` React Query, che si occupa anche di aggiornare la cache in seguito all'operazione.

Gli adapter svolgono una funzione di traduzione: si occupano della conversione delle date e degli intervalli orari, del mapping dei tipi di prenotazione nei valori interni utilizzati dall'applicazione, dell'arricchimento dei dati con flag calcolati (ad esempio per indicare se una prenotazione può ancora essere modificata) e della gestione delle eventuali discrepanze tra la rappresentazione prevista dall'API e quella richiesta dalla UI.

Grazie a questo approccio, tutti i moduli dell'applicazione sono stati completamente integrati con il client TypeScript generato dalla specifica OpenAPI. Il prototipo basato su dati mock è stato progressivamente sostituito da un'architettura basata sulle chiamate API, in cui ogni flusso è gestito da hook tipizzati di React Query e da funzioni di conversione dedicate. Questa metodologia consente di mantenere un allineamento costante tra il modello definito nel capitolo precedente e le esigenze operative dell'interfaccia utente.

5.3 Gestione dei dati mock e validazione con il frontend

Durante questa fase, l'applicazione *PoliTO Faculty* ha continuato a basarsi su dati mock, ma con una differenza sostanziale rispetto al prototipo iniziale: i mock sono stati allineati in modo esplicito alla struttura definita nella specifica OpenAPI.

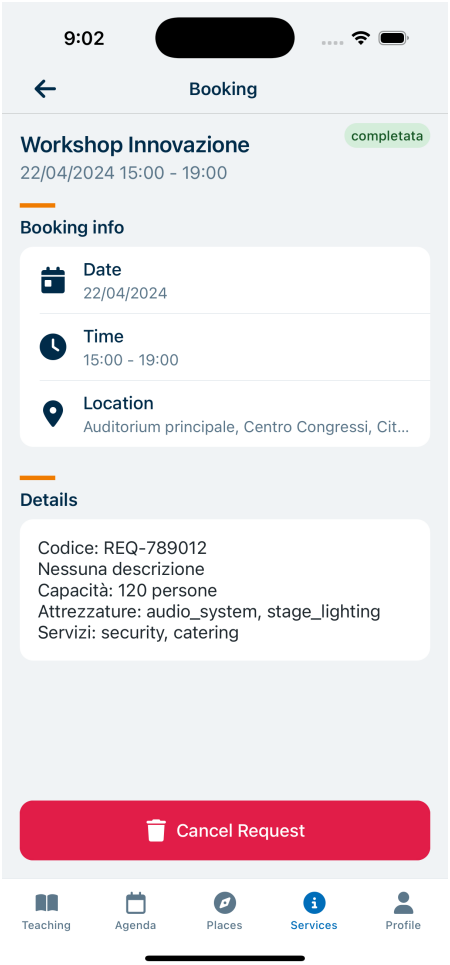

Figura 5.6: Dettaglio di una prenotazione con l'azione di cancellazione integrata via API.

Nel repository *api-spec* è stato introdotto il file `openapi-examples.yaml`, che raccoglie payload di esempio coerenti con gli schemi dei vari domini (Common, Students, Faculty). Questo file svolge un duplice ruolo:

- fornire esempi da utilizzare come riferimento per i mock, così che i dati simulati rispettino i vincoli di struttura, tipi e campi obbligatori definiti dalla specifica;
- costituire una base per la documentazione e per i test manuali, permettendo di verificare rapidamente se le schermate dell'app riescono a visualizzare correttamente tutte le informazioni utilizzando la struttura definita nella specifica.

La validazione con il frontend avviene quindi su due livelli:

1. **Validazione di compilazione**, garantita dai tipi generati nel client TypeScript: eventuali discrepanze tra modelli usati dall'applicazione e schemi definiti in OpenAPI emergono come errori di tipo in fase di sviluppo.
2. **Validazione visiva e di flusso**, ottenuta caricando nelle schermate dati mock costruiti a partire dagli esempi della specifica e verificando che le interazioni (liste, dettagli, azioni di aggiornamento) risultino coerenti con le aspettative dei docenti.

Esempi di adattamento dell'interfaccia

L'integrazione con le API reali ha evidenziato che alcuni flussi del prototipo non erano perfettamente compatibili con quelli previsti dal contratto.

Un caso significativo è quello della firma digitale: nella prima versione del frontend il flusso era modellato come un'unica azione di conferma, mentre le API del dominio *Faculty* espongono un processo composto da due step distinti (verifica dell'OTP e conferma tramite PIN). Per allineare l'interfaccia al comportamento del backend è stato necessario introdurre due schermate dedicate, rispettivamente per la richiesta dell'OTP e per l'inserimento del PIN, e aggiornare gli hook di integrazione per gestire i diversi stati restituiti dalle chiamate (invio del codice, errori di validazione, sessioni scadute). Le schermate mostrate nelle Figure 5.4 e 5.5 sono il risultato di questo adattamento.

Un intervento più contenuto ha riguardato l'introduzione di una sezione *News* per i docenti, ottenuta portando nell'applicazione *PoliTO Faculty* una funzionalità già presente nell'applicazione dedicata agli studenti. In questo caso è stato sufficiente definire nuovi hook React Query che riutilizzano le API del dominio comune e connetterli a componenti di lista esistenti, seguendo gli stessi pattern di conversione adottati per gli altri moduli.

5.4 Controllo della qualità del codice

L'integrazione con il client TypeScript generato si appoggia su una serie di strumenti pensati per mantenere omogeneo e manutenibile il codice dell'applicazione. A livello di progetto vengono utilizzati ESLint e Prettier, affiancati dal type checking di TypeScript in modalità *strict* [20, 21]. La configurazione di ESLint estende le regole ufficiali per React Native e introduce controlli specifici sull'uso di TypeScript, sulla gestione degli import e sull'utilizzo della console, mentre Prettier si occupa della formattazione automatica e dell'ordinamento coerente degli import.

Gli script `npm` dedicati (`lint`, `format`, `typecheck`) consentono di eseguire in modo ripetibile le verifiche di linting, formattazione e type checking, con una politica di tolleranza zero rispetto ai warning di ESLint. Queste verifiche vengono applicate in più punti: localmente tramite *pre-commit hook* (gestiti con Husky e `lint-staged`, che eseguono auto-fix e formattazione sui file modificati), in fase di *pre-push* attraverso lo script `npm run check` e infine in CI, dove un workflow GitHub dedicato esegue in parallelo *lint*, *format check* e *type checking* su ogni *pull request* [22, 23].

In parallelo ai controlli sintattici e di stile, il progetto adotta alcune convenzioni architetturali legate all'uso del client API generato. Le chiamate verso le API sono concentrate negli hook React Query definiti in `src/core/queries/`, che fungono da unico punto di accesso ai client; i tipi esposti dal client non vengono propagati direttamente ai componenti, ma attraversano funzioni di conversione dedicate che mappano le entità delle API in modelli interni all'applicazione. Questa combinazione di regole di stile, verifica statica e convenzioni di progetto contribuisce a mantenere chiaro il confine tra integrazione con il backend e logica di presentazione, riducendo la possibilità di introdurre code smell o dipendenze non desiderate.

Dal punto di vista dell'applicazione, questo approccio ha due effetti concreti:

1. ogni versione del client TypeScript è riconducibile a una versione specifica della PoliTO API, semplificando il tracciamento delle dipendenze;

2. l'aggiornamento del client nel progetto *PoliTO Faculty* avviene partendo da una specifica già validata, riducendo il rischio di dover gestire errori strutturali direttamente nel codice dell'app.

Nel complesso, il lavoro descritto in questo capitolo ha permesso di passare da un prototipo basato esclusivamente su dati mock a un'applicazione *API-first*, in cui la specifica OpenAPI, il client TypeScript generato, gli hook React Query e la pipeline di validazione costituiscono un'unica catena coerente.

Su queste basi è stata impostata la fase successiva di verifica tecnica: il capitolo seguente descrive i test manuali e le prove automatizzate condotte sui casi

d'uso prioritari, con l'obiettivo di valutare in modo sistematico la correttezza del comportamento osservato rispetto al contratto definito dalle API.

Capitolo 6

Verifica Tecnica dell'Integrazione

Il lavoro descritto nei capitoli precedenti ha portato alla definizione di una specifica OpenAPI aggiornata, alla generazione di un client TypeScript e alla loro integrazione all'interno del prototipo mobile *PoliTO Faculty*. Questo capitolo descrive come tale integrazione sia stata verificata dal punto di vista tecnico, combinando prove manuali sull'applicazione con una serie di test automatici sui componenti chiave del codice.

6.1 Obiettivi della verifica tecnica

La verifica è stata svolta sul prototipo *PoliTO Faculty*, collegato a un backend mock basato su *Prism* [17]. Non è stato quindi utilizzato un ambiente di Ateneo reale, ma un server che restituisce risposte coerenti con il bundle OpenAPI descritto nel capitolo precedente.

L'obiettivo principale della verifica tecnica era:

- dimostrare che il *client TypeScript* generato dalla specifica OpenAPI fosse effettivamente utilizzabile dall'applicazione, senza discrepanze strutturali tra i tipi generati e i payload scambiati con il backend;
- verificare che l'applicazione fosse in grado di rappresentare correttamente, a livello di interfaccia utente, le risposte delle API per i principali casi d'uso lato docente, mantenendo coerenza con i modelli introdotti nella specifica.

L'attenzione è stata rivolta in particolare ai flussi che caratterizzano l'utilizzo quotidiano dell'app da parte di un docente: autenticazione e login, consultazione del profilo, gestione degli incarichi didattici e degli esami, modulo di firma digitale,

prenotazioni di spazi, consultazione di news ed emergenze, ricerca persone. Per ciascuna di queste aree, l'obiettivo minimo era che almeno un percorso *happy path* potesse essere completato senza errori, utilizzando esclusivamente le API definite nella nuova specifica.

6.2 Metodologia di validazione

La verifica tecnica è stata condotta adottando una combinazione di prove manuali sull'applicazione in esecuzione e test automatici sui componenti di integrazione con le API. In questa sezione vengono descritti l'ambiente di test, il modo in cui sono stati eseguiti i flussi manuali e il perimetro dei test automatici implementati.

Scelte di validazione e perimetro dei test

Le attività di verifica tecnica sono state impostate con l'obiettivo di dimostrare la fattibilità dell'integrazione tra la specifica OpenAPI aggiornata, il client TypeScript generato e il prototipo mobile *PoliTO Faculty*. In altre parole, la validazione è stata svolta per verificare che i flussi principali potessero essere eseguiti end-to-end dal punto di vista dell'applicazione, più che a certificare la prontezza del sistema per un rilascio in produzione.

In questa prospettiva, sono stati privilegiati test funzionali condotti sugli emulatori, utilizzando come backend un server mock basato su *Prism* configurato direttamente a partire dalla specifica. Questa scelta ha permesso di concentrare l'attenzione sulla coerenza del contratto e sulla correttezza dell'integrazione lato client, riducendo i tempi necessari per allestire e mantenere un ambiente backend reale, che sarebbe stato fuori scala rispetto alle risorse disponibili per una tesi magistrale.

Allo stesso tempo, sono state escluse dal perimetro della validazione alcune tipologie di test che, pur rilevanti in un contesto di produzione, avrebbero richiesto uno sforzo difficilmente compatibile con lo scopo di questo lavoro. In particolare, non sono stati eseguiti test di carico o di performance sulle API, né sono state condotte verifiche specifiche sugli aspetti di sicurezza applicativa. Anche l'automazione completa di scenari end-to-end su emulatori è stata considerata fuori dal perimetro, privilegiando invece una combinazione di test manuali sui flussi critici e test automatici mirati sui componenti di integrazione più sensibili (come hook e converter).

Nel complesso, le scelte di validazione riflettono quindi un compromesso consapevole: da un lato garantire un livello di confidenza adeguato sulla correttezza dell'integrazione tra specifica, client e prototipo mobile; dall'altro mantenere il focus sul contributo principale della tesi, evitando di disperdere le risorse su attività che appartengono più propriamente a una fase di industrializzazione del sistema.

6.2.1 Ambiente di test e strumenti

Dal punto di vista infrastrutturale, l'ambiente di verifica era composto da:

- il repository *api-spec*, configurato per generare il bundle `openapi.yaml` a partire da `src/index.yaml` e per avviare un server mock basato su *Prism*, in grado di rispondere alle richieste secondo gli schemi e gli esempi definiti nella specifica;
- il repository dell'applicazione *PoliTO Faculty*, in esecuzione in modalità di sviluppo su emulatore, configurata per utilizzare il client TypeScript generato come unico punto di accesso alle API;
- gli strumenti di supporto allo sviluppo già introdotti nel capitolo precedente (`script npm`, React Native, React Query, TypeScript in modalità *strict*).

L'esecuzione sull'emulatore, in combinazione con il server mock, ha permesso di iterare rapidamente sulle modifiche alla specifica e alla logica di integrazione, sfruttando le funzionalità di *Fast Refresh* per osservare quasi in tempo reale gli effetti dei cambiamenti [6].

6.2.2 Verifica manuale dei flussi applicativi

La parte principale della verifica è stata svolta in modo manuale, eseguendo sull'emulatore flussi d'uso realistici per ciascuna sezione dell'applicazione. Non sono state predisposte checklist formali o casi di test dettagliati, ma per ogni area funzionale è stato seguito un percorso rappresentativo di come un utente utilizzerebbe l'app:

- accesso all'applicazione tramite schermata di login, con verifica del corretto instradamento verso le viste protette;
- consultazione del profilo docente e delle informazioni di contatto;
- navigazione nella sezione *Teaching*, con visualizzazione della lista corsi e degli appelli d'esame, accesso alle liste di studenti e collaboratori;
- utilizzo del modulo di firma digitale, dalla lista dei documenti da firmare all'inserimento di OTP e PIN;
- visualizzazione e gestione delle prenotazioni di spazi associati al docente;
- consultazione di news ed emergenze, incluse la lista e il dettaglio delle segnalazioni;

- esecuzione di ricerche di persone, per verificare la corretta integrazione delle nuove API di directory.

Per ciascuno di questi flussi è stato verificato che:

- i dati visualizzati fossero coerenti con la struttura delle risposte REST simulate dal server mock;
- non emergessero errori applicativi o stati incoerenti nel passaggio tra le schermate;
- l'interfaccia mantenesse lo stesso comportamento definito dal prototipo iniziale.

Questo tipo di verifica ha permesso di individuare rapidamente eventuali discrepanze tra la specifica OpenAPI e le esigenze concrete della UI, oltre a fornire una prima conferma della solidità del modello *API-first* adottato.

6.2.3 Test automatici su hook e componenti di integrazione

Oltre alla verifica manuale, il progetto prevede anche una serie di test automatici, concentrati in particolare sugli hook di React Query e sulle funzioni di supporto utilizzate per integrare il client TypeScript all'interno dell'applicazione.

I test unitari sono stati implementati con *Jest* e organizzati principalmente in due categorie [24]:

- test sui *converter* e sulle utilità di trasformazione dei dati, ad esempio per il dominio delle segnalazioni di guasto, dove si verifica che le funzioni di conversione formattino correttamente le date e gestiscano in modo robusto timestamp mancanti o non validi;
- test sugli hook React Query dedicati ai vari domini (autenticazione, profilo, corsi, esami, calendario, emergenze, fault, prenotazioni di spazi, news, persone, studenti), che controllano la creazione dei client specifici, la stabilità delle chiavi di query e il corretto funzionamento degli helper di cache e di risoluzione delle risposte.

Il criterio di successo per questa parte automatizzata era che tutti i test unitari risultassero superati, integrandosi con i controlli di *linting*, formattazione e *type checking* descritti nel capitolo precedente.

6.3 Casi d'uso verificati

In questa sezione vengono sintetizzati i principali casi d'uso verificati durante la fase di test, sia dal punto di vista della navigazione manuale sull'applicazione, sia in relazione ai test automatici sui componenti di integrazione.

6.3.1 Autenticazione e login

Il primo flusso verificato riguarda l'accesso all'applicazione. È stata introdotta una schermata di login dedicata, assente nel prototipo iniziale, che permette al docente di autenticarsi prima di accedere alle sezioni protette. La verifica ha riguardato:

- la corretta interazione con le API di autenticazione, tramite il client TypeScript generato;
- la propagazione dello stato di autenticazione nel contesto dell'app;
- il passaggio dalla schermata di login alla pagina principale dell'applicazione senza errori.

6.3.2 Profilo docente

Per il profilo, la verifica ha interessato la sezione che espone i dati anagrafici e amministrativi del docente.

Il flusso testato prevede:

- l'accesso alla schermata di profilo;
- il caricamento dei dati dal backend mock tramite l'hook dedicato e il client del dominio *Faculty*;
- la visualizzazione e, quando previsto, l'aggiornamento dei dati anagrafici e amministrativi, delle pubblicazioni e dei di corsi in gestione;

L'obiettivo era verificare che lo schema del profilo definito nella specifica OpenAPI fosse sufficiente a popolare la vista così come modellata nel prototipo.

6.3.3 Incarichi didattici, corsi ed esami

La sezione *Teaching* riunisce le funzionalità legate agli incarichi didattici e alla gestione degli esami.

Per i corsi sono stati verificati:

- il caricamento della lista degli insegnamenti associati al docente;
- l'apertura del dettaglio di un singolo corso, con accesso alle liste di studenti e collaboratori;
- la gestione degli avvisi del corso, del materiale didattico, del registro lezioni e degli elaborati;

- la coerenza tra i dati restituiti dalle API e quelli visualizzati (titolo del corso, anno accademico, numero di iscritti, ruoli dei collaboratori).

Per gli esami, la verifica ha riguardato:

- la lista degli appelli d'esame lato docente;
- il dettaglio di un appello, con elenco degli studenti iscritti e informazioni principali (data, orario, aula);
- l'utilizzo delle API specifiche per la notifica agli iscritti.

In entrambi i casi, lo scopo era dimostrare che gli hook React Query e i client dei domini *Courses* ed *Exams* fossero in grado di sostenere i flussi previsti, utilizzando i modelli definiti nella specifica.

6.3.4 Flusso di firma digitale

Il modulo di firma digitale rappresenta uno dei casi d'uso più delicati. La verifica ha riguardato due aspetti complementari:

- la lista dei documenti, suddivisi tra documenti ancora da firmare e documenti già firmati;
- il flusso di firma vero e proprio, modellato come processo a due step (OTP e PIN) in linea con le nuove API.

Partendo dalla schermata di visualizzazione della lista di documenti, è stato verificato che:

- le chiamate agli endpoint `/faculty/signing/documents` restituissero un insieme di documenti coerente con le aspettative della UI;
- la selezione di un documento conducesse correttamente alle schermate di inserimento OTP e PIN, collegate alle rispettive mutazioni React Query;
- la sequenza OTP $\rightarrow$ PIN potesse essere completata senza errori, con aggiornamento dello stato del documento nella lista.

Questo caso d'uso ha confermato la capacità del prototipo di riflettere in modo fedele il flusso di firma digitale descritto nella specifica OpenAPI, dopo l'adattamento dell'interfaccia rispetto al prototipo originale a singolo step.

6.3.5 Prenotazioni di spazi

Per le prenotazioni, la verifica si è concentrata sulla consultazione e gestione delle richieste associate al docente. Il flusso testato comprende:

- la visualizzazione della lista delle prenotazioni, alimentata dagli hook che interrogano le corrette API;
- l'apertura del dettaglio di una prenotazione, con controllo di tutti i campi visualizzati (intervallo temporale, luogo, capacità, attrezzature, servizi);
- l'esecuzione di operazioni di aggiornamento o cancellazione, con verifica del corretto aggiornamento della lista in seguito alle mutation.

La presenza di un adattatore dedicato per il dominio delle prenotazioni ha reso possibile mantenere un modello interno coerente con la UI, pur in presenza di schemi API relativamente articolati.

6.3.6 News ed emergenze

Sono stati testati anche i flussi di consultazione di news ed emergenze, che riutilizzano in parte API già presenti per l'applicazione rivolta agli studenti.

Per quanto riguarda le news, il flusso di verifica ha considerato i seguenti aspetti:

- il caricamento della lista di news tramite le API;
- l'apertura del dettaglio di una singola notizia.

Per le emergenze, sono stati verificati:

- la lista delle emergenze istituzionali, basata sugli endpoint `/emergencies`;
- il dettaglio di una singola emergenza, con controllo dei campi principali (titolo, descrizione, data, eventuali indicazioni operative).

Questi flussi hanno permesso di valutare la capacità della specifica estesa di coprire casi d'uso trasversali, non legati esclusivamente ai servizi tipici della didattica.

6.3.7 Ricerca persone

Infine, è stato testato il flusso di ricerca persone, che consente al docente di individuare membri della comunità accademica a partire da una stringa di ricerca.

La verifica ha riguardato:

- l'inserimento di una query nella schermata di ricerca;

- il caricamento dei risultati, con elenco di persone corrispondenti;
- la visualizzazione di informazioni sintetiche (nome, struttura, ruolo), utili per contestualizzare i risultati.

Questo caso d'uso è stato uno dei punti in cui è stato necessario intervenire sugli schemi delle API, come descritto nella sezione successiva.

6.3.8 Ambito coperto dai test automatici

I test automatici hanno affiancato la verifica manuale coprendo soprattutto:

- le funzioni di conversione dei dati, ad esempio per le segnalazioni di guasto, dove è importante garantire una formattazione consistente delle date e una gestione robusta dei valori mancanti;
- gli hook React Query dei principali domini applicativi (autenticazione, profilo, corsi, esami, calendario, emergenze, guasti, prenotazioni, news, persone, studenti), verificando la creazione dei client corretti, le chiavi di query e il funzionamento degli *helper* di cache.

Questa copertura, pur non esaustiva, ha fornito un ulteriore livello di confidenza sulla stabilità dell'integrazione, soprattutto in presenza di cambiamenti incrementali della specifica.

6.4 Risultati e osservazioni

La combinazione di prove manuali e test automatici ha prodotto una serie di risultati che possono essere sintetizzati su due livelli: esito complessivo dei flussi e osservazioni su incongruenze emerse tra specifica e implementazione.

6.4.1 Esito complessivo

Dal punto di vista funzionale, tutti i flussi considerati prioritari per i docenti sono stati portati a termine con successo sull'emulatore, utilizzando il server mock basato sulla specifica OpenAPI estesa.

In particolare: la sequenza login → sezioni *Teaching*, *Agenda*, *Mappe*, *Servizi*, e *Profilo* è risultata percorribile senza errori bloccanti.

Sul fronte dei test automatici, tutti i test unitari Jest implementati sui converter e sugli hook risultano superati. Questo indica che le assunzioni fatte in fase di integrazione (creazione dei client, chiavi di cache, mapping delle risposte) sono compatibili con la specifica corrente.

6.4.2 Incongruenze tra API e UI emerse durante i test

La verifica ha portato alla luce alcune incongruenze tra quanto definito inizialmente nella specifica OpenAPI e quanto richiesto, in pratica, dall'interfaccia utente.

Un primo caso ha riguardato gli eventi a calendario: durante la consultazione dell'agenda docente si è evidenziato che alcuni campi necessari alla vista non erano presenti nella risposta degli endpoint del dominio *Faculty Calendars*. Per risolvere la discrepanza è stato necessario estendere gli schemi relativi agli eventi, aggiungendo i campi mancanti direttamente nella specifica OpenAPI e rigenerando il client TypeScript, in modo che i converter potessero lavorare su un modello completo.

Un secondo caso ha coinvolto il flusso di ricerca persone. La UI prevedeva la visualizzazione del *role name* per ciascun risultato, ma la risposta delle API di directory non includeva inizialmente questa informazione. Anche in questo scenario la soluzione ha richiesto un aggiornamento della specifica: il campo relativo al ruolo è stato aggiunto allo schema della persona, con conseguente rigenerazione del client e aggiornamento del mapping lato frontend.

Questi episodi mostrano come la verifica tecnica, pur svolta su un prototipo collegato a un backend mock, abbia avuto un impatto diretto sulla qualità della specifica, colmando alcune lacune del contratto.

6.4.3 Impatto sulla robustezza dell'integrazione

Nel complesso, i risultati ottenuti suggeriscono che l'approccio adottato, basato su una specifica OpenAPI evoluta, sulla generazione del client TypeScript e su una serie di convenzioni per l'uso degli hook React Query, è adeguato a sostenere i flussi principali della *PoliTO Faculty* app.

La presenza di test unitari su converter e hook ha contribuito a mantenere stabili le interfacce interne quando gli schemi venivano aggiornati, riducendo il rischio di regressioni nella gestione dei dati.

6.5 Limitazioni della validazione tecnica

La verifica descritta in questo capitolo presenta alcuni limiti intrinseci, legati sia alle scelte metodologiche sia ai vincoli pratici del progetto. È importante esplicitarli per delimitare il perimetro dei risultati ottenuti e chiarire quali aspetti richiederanno ulteriori approfondimenti in fasi successive.

6.5.1 Assenza di un backend reale

Tutti i test sono stati condotti utilizzando un server mock basato su *Prism*, configurato a partire dal bundle OpenAPI. Questo ha permesso di concentrarsi sulla

coerenza tra specifica, client e applicazione, ma ha escluso dalla verifica:

- l'integrazione con sistemi esterni reali e con i dati effettivamente presenti nei sistemi di Ateneo;
- la gestione di casi limite e dati anomali che emergono tipicamente solo in ambienti di produzione o pre-produzione;
- eventuali vincoli infrastrutturali, come latenza, limiti di rate o altre restrizioni, che potrebbero influire sul comportamento dell'applicazione.

L'assenza di un ambiente di *staging* reale è quindi un limite esplicito: l'obiettivo della verifica non era certificare il comportamento dell'intero ecosistema, ma valutare se la specifica OpenAPI fosse sufficiente a sostenere i flussi del prototipo [1].

6.5.2 Performance e carichi reali

Non sono stati eseguiti test di performance o di carico. In particolare:

- non sono state simulate richieste concorrenti da parte di più utenti;
- non sono stati misurati tempi di risposta in scenari di stress;
- non è stata analizzata in dettaglio l'efficienza delle strategie di caching lato client in condizioni di uso intensivo.

Questi aspetti esulano dall'ambito del lavoro svolto, che si è concentrato sulla correttezza funzionale dei flussi e sulla coerenza tra modelli di dominio, specifica e interfaccia.

6.5.3 Sicurezza applicativa

La validazione tecnica non ha incluso test specifici di sicurezza, come analisi di vulnerabilità, *penetration test* o simulazioni di attacchi mirati. Gli aspetti di sicurezza sono stati considerati principalmente in fase di progettazione, ma non hanno costituito una fase di verifica autonoma.

In una prospettiva futura, una valutazione di sicurezza completa richiederebbe la disponibilità di un backend reale e il coinvolgimento dei team responsabili della sicurezza applicativa e infrastrutturale.

6.5.4 Copertura dei test automatici

Infine, anche la copertura dei test automatici presenta dei limiti. I test unitari implementati mirano a verificare il comportamento di singole funzioni (converter, hook, helper di cache), senza coprire l'intero spettro dei flussi utente end-to-end.

Questa copertura è adeguata rispetto all'obiettivo di verificare la solidità dell'integrazione in un contesto di prototipo, ma non sostituisce una suite completa di test di integrazione e di sistema, che sarà necessaria in vista di un eventuale rilascio in ambienti più vicini alla produzione.

In sintesi, la validazione tecnica ha confermato che la specifica OpenAPI aggiornata e il client TypeScript generato sono adeguati a sostenere i principali flussi previsti dal prototipo *PoliTO Faculty*, consentendo al tempo stesso di affinare il contratto API e correggere alcune lacune emerse durante i test. L'attività svolta ha inoltre messo in luce ambiti, come performance, sicurezza e verifiche su un backend reale, che richiederanno interventi dedicati nelle fasi successive del progetto.

Capitolo 7

Conclusioni e Sviluppi Futuri

L'obiettivo generale di questa tesi era la progettazione e integrazione di un livello API robusto, documentato e verificabile per *PoliTO Faculty*, partendo dai bisogni emersi rispetto all'applicazione esistente. Nei capitoli centrali il lavoro si è concentrato sugli aspetti tecnici: evoluzione della specifica *PoliTO API*, generazione del client TypeScript e integrazione all'interno del prototipo *mobile* dedicato ai docenti. In questo capitolo vengono riassunti i risultati ottenuti, l'impatto sul progetto *PoliTO Faculty*, i principali limiti del lavoro svolto e le possibili direzioni di sviluppo futuro.

7.1 Sintesi dei risultati ottenuti

Dal punto di vista tecnico, i risultati della tesi possono essere ricondotti a tre punti principali: la ristrutturazione della specifica *PoliTO API*, l'introduzione del dominio *Faculty* e l'integrazione del client generato nel prototipo *mobile*.

Per quanto riguarda la specifica è stata riorganizzata passando da un file monolitico a una struttura OpenAPI modulare, suddivisa per domini (Common, Students, Faculty) e composta da componenti riusabili. L'introduzione di schemi condivisi, parametri e risposte centralizzate ha reso più semplice estendere le API e mantenere coerenza tra i vari moduli. La configurazione degli script `npm` per il bundling, la validazione e la generazione del client ha inoltre reso più agevole il processo di aggiornamento della specifica.

Parallelamente, è stato definito e documentato un insieme di API dedicate ai docenti. Il dominio *Faculty* copre i principali casi d'uso identificati per *PoliTO Faculty*: consultazione e aggiornamento del profilo, gestione di corsi ed esami lato

docente, flusso di firma digitale, consultazione di calendari ed eventi, integrazione con emergenze e prenotazioni già modellate nel dominio comune.

Infine, la specifica è stata collegata a un client TypeScript generato automaticamente e integrata nel prototipo *mobile*. L'applicazione, inizialmente basata su dati mock e priva di una logica di accesso alle API, è stata progressivamente adattata ad utilizzare le API: le chiamate verso il backend passano attraverso hook React Query tipizzati, funzioni di conversione tra tipi API e modelli interni, e un insieme di pattern condivisi per la gestione di cache, errori e stati di caricamento. Il risultato è un codice più pulito, leggibile e mantenibile, pronto a dialogare con un backend reale non appena sarà disponibile.

Nel complesso, il lavoro svolto dimostra che è possibile progettare e integrare un livello API consistente per *PoliTO Faculty*, mantenendo allineati specifica, client e prototipo *mobile* e ponendo le basi per sviluppi successivi.

7.2 Impatto tecnico sul progetto *PoliTO Faculty*

L'impatto tecnico sul progetto *PoliTO Faculty* è significativo, sia lato API sia lato applicazione *mobile*.

Per quanto riguarda il progetto *api-spec*, la nuova struttura modulare rende più agevole intervenire sulla specifica senza dover manipolare un unico file di grandi dimensioni. La suddivisione per domini e l'introduzione di componenti riusabili riducono il rischio di incoerenze tra schemi simili. La presenza di script `npm` per il bundling e la validazione, insieme ai workflow GitHub dedicati, consente di inserire la specifica all'interno di un flusso di integrazione continua, in cui ogni modifica è verificata prima di essere resa disponibile per la generazione del client.

Dal punto di vista dell'applicazione *PoliTO Faculty*, l'impatto principale riguarda l'organizzazione del codice e la gestione dell'integrazione con il backend. L'introduzione di un *layer* di hook React Query per dominio e l'uso di converter per separare i tipi API dai modelli interni hanno reso il codice più uniforme e strutturato. I diversi flussi (profilo, corsi, esami, firma digitale, prenotazioni, emergenze, news, ricerca persone) seguono la stessa impostazione: il componente UI si appoggia a un hook, l'hook utilizza il client generato e i converter si occupano di adattare le risposte alle esigenze dell'interfaccia.

Questa coerenza architetturale facilita il debugging, riduce la probabilità di introdurre regressioni quando la specifica cambia e rende più semplice aggiungere nuove funzionalità basate su API già esistenti. Pur rimanendo confinato a un prototipo e a un backend mock, il lavoro ha quindi contribuito a chiarire come dovrebbe essere strutturata l'integrazione tecnica di un'app faculty in un ecosistema già popolato da altri client (come l'app studenti).

7.3 Limiti e margini di miglioramento

Il percorso seguito presenta alcuni limiti che è opportuno esplicitare, sia per delimitare il perimetro dei risultati, sia per indicare dove il lavoro potrebbe essere esteso o rafforzato.

Un primo limite riguarda la misurazione degli effetti delle modifiche introdotte. Il lavoro si è concentrato su una ristrutturazione architetturale e su un'integrazione tecnica, senza definire in modo sistematico indicatori quantitativi. Di conseguenza, molti benefici sono stati valutati in termini qualitativi (maggiore leggibilità del codice, riduzione dei conflitti, facilità di estensione), senza una metrica numerica che ne quantifichi l'impatto.

Un secondo limite riguarda il processo di testing. I test descritti nel capitolo precedente sono stati efficaci per individuare discrepanze tra specifica e implementazione e per verificare il corretto funzionamento degli hook e dei converter, ma non sono stati guidati da un piano di test formalizzato.

Un terzo limite, già evidenziato nella verifica tecnica, riguarda l'uso di un backend mock eseguito in locale. L'intero lavoro di integrazione è stato validato contro un server basato sulla specifica OpenAPI, senza interazione con API reali e senza accesso a dati di produzione o pre-produzione. Questo approccio ha permesso di concentrarsi sulla coerenza tra contratto e codice, ma non consente di trarre conclusioni su aspetti come latenza, prestazioni sotto carico o comportamento in scenari di errore reali, né di valutare in modo approfondito i profili di sicurezza.

Dal punto di vista dei margini di miglioramento, l'area più evidente è proprio il processo di testing. Una fase successiva potrebbe prevedere la definizione di casi di test scritti, allineati ai requisiti applicativi, con indicazione esplicita dei passi da eseguire, dei dati di partenza e dei risultati attesi. Su questa base sarebbe possibile estendere la suite automatizzata, passando da test prevalentemente unitari a test di integrazione e, dove possibile, a test end-to-end che attraversino l'intero stack dall'API al componente UI. Una struttura di questo tipo renderebbe più robusta l'evoluzione della specifica e del prototipo nel tempo.

7.4 Sviluppi futuri e possibili estensioni del sistema

Gli sviluppi futuri si collocano idealmente lungo tre direttrici: l'attivazione di un backend reale, il riuso dei componenti tra le app esistenti e l'estensione delle integrazioni verso piattaforme esterne.

La prima direttrice riguarda il passaggio da un server mock a un ambiente reale (o di *staging*) per la *PoliTO API*. Una volta che le API modellate nella specifica saranno implementate, il client TypeScript e il prototipo *PoliTO Faculty* potranno

essere collegati a un backend effettivo, consentendo di verificare il comportamento dell'applicazione con dati e vincoli infrastrutturali reali. Questo passaggio permetterebbe di valutare in modo più concreto le prestazioni, la gestione degli errori, l'impatto della latenza di rete e, più in generale, la maturità della soluzione in vista di un eventuale rilascio.

La seconda direttrice riguarda l'allineamento tra l'app studenti e l'app faculty. Un'evoluzione naturale del lavoro svolto sarebbe la definizione di una libreria UI condivisa tra i due progetti, costruita a partire dai pattern già consolidati nel prototipo (liste, schermate di dettaglio, componenti per le card di corsi, esami, eventi, news, emergenze). Una libreria di questo tipo contribuirebbe a mantenere coerenza visiva e di comportamento tra le applicazioni, riducendo la duplicazione di codice e semplificando l'introduzione di nuove funzionalità comuni.

La terza direttrice riguarda il completamento delle integrazioni previste con piattaforme esterne. Nel lavoro attuale non sono state implementate, ad esempio, le integrazioni con sistemi come Outlook, che sono però rilevanti per il modo in cui i docenti organizzano la propria attività quotidiana. La specifica e il prototipo potrebbero essere estesi per coprire meglio questi scenari, sfruttando i punti di aggancio già presenti nell'ecosistema *PoliTO API* e mantenendo coerente il modello dati tra applicazioni interne e servizi esterni.

Queste direzioni indicano alcune attività concrete che potrebbero proseguire il lavoro svolto, contribuendo a trasformare il prototipo in un'applicazione matura e idonea a un rilascio pubblico.

7.5 Considerazioni finali

Il percorso affrontato in questa tesi mostra come, in un contesto già articolato come quello dei servizi digitali di Ateneo, l'introduzione di un nuovo dominio applicativo richieda prima di tutto un lavoro di progettazione sul livello API. La ristrutturazione della specifica *PoliTO API*, l'introduzione del dominio *Faculty* e l'integrazione con il prototipo *mobile* hanno evidenziato che la qualità del contratto tra backend e frontend incide direttamente sulla semplicità con cui è possibile sviluppare, testare e mantenere le funzionalità lato app.

L'esperienza maturata conferma l'importanza di definire una specifica chiara e coerente prima di investire in modo esteso sullo sviluppo di nuove interfacce. Un modello *API-first* limita la duplicazione e rende più prevedibili gli effetti delle modifiche. Allo stesso tempo, l'uso di pattern consistenti lato applicazione (hook React Query, converter, gestione centralizzata delle chiavi di cache) contribuisce a tenere separati i ruoli: le API espongono servizi e dati, l'applicazione si occupa di presentarli in modo efficace.

Pur nei limiti dovuti all'uso di un backend mock e all'assenza di misure quantitative, il lavoro svolto dimostra che *PoliTO Faculty* può appoggiarsi a un livello API solido e che i principali casi d'uso dei docenti trovano una rappresentazione tecnica plausibile all'interno dell'ecosistema esistente. I capitoli dedicati alla progettazione, all'implementazione e alla verifica tecnica forniscono una base su cui costruire ulteriori sviluppi, dalla messa in produzione di un backend reale all'estensione delle funzionalità esposte.

Bibliografia

[1] Ian Sommerville. *Software Engineering*. 10ª ed. London: Pearson, 2015 (cit. alle pp. 2, 10, 51).

[2] OpenAPI Initiative. *OpenAPI Specification 3.1.1*. `https://spec.openapis.org/oas/v3.1.1.html`. 2024. (Visitato il giorno 03/11/2025) (cit. alle pp. 2, 5, 15).

[3] *ISO 9241-210:2019 – Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems*. Geneva, Switzerland: International Organization for Standardization, 2019. URL: `https://www.iso.org/standard/77520.html` (cit. alle pp. 5, 13).

[4] Mehdi Medjaoui, Erik Wilde, Ronnie Mitra e Mike Amundsen. *Continuous API Management: Making the Right Decisions in an Evolving Landscape*. Sebastopol, CA: O'Reilly Media, 2018 (cit. a p. 5).

[5] Meta Platforms. *Setting up the development environment*. React Native Documentation. Accessed: 2025-11-18. 2024. URL: `https://reactnative.dev/docs/environment-setup` (cit. a p. 5).

[6] Meta Platforms. *React Native Documentation*. Accessed: 2025-11-18. 2024. URL: `https://reactnative.dev/docs/getting-started` (cit. alle pp. 5, 27, 44).

[7] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang et al. «A Survey of Large Language Models». In: *arXiv preprint arXiv:2303.18223* (2023) (cit. a p. 8).

[8] Julian Ashwin, Aditya Chhabra e Vijayendra Rao. *Using Large Language Models for Qualitative Analysis can Introduce Serious Bias*. Rapp. tecn. 10597. World Bank Policy Research Working Paper, 2023 (cit. a p. 9).

[9] Andrew Kirkpatrick, Joshue O'Connor, Alastair Campbell e Michael Cooper. *Web Content Accessibility Guidelines (WCAG) 2.2*. W3C Recommendation. World Wide Web Consortium (W3C), 2023. URL: `https://www.w3.org/TR/WCAG22/` (cit. a p. 13).

[10] Politecnico di Torino. *PoliTO Students App*. `https://github.com/polito/students-app`. Repository GitHub open source. 2024 (cit. a p. 14).

[11] Roy Thomas Fielding. «Architectural styles and the design of network-based software architectures». Tesi di dott. University of California, Irvine, 2000 (cit. a p. 19).

[12] Mark Massé. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc.", 2011 (cit. a p. 19).

[13] Michael Jones e Dick Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. Internet Engineering Task Force (IETF), 2012. URL: `https://tools.ietf.org/html/rfc6750` (cit. a p. 21).

[14] OpenAPI Generator Contributors. *OpenAPI Generator: One-stop solution for modern API client generation*. Accessed: 2025-11-03. 2024. URL: `https://openapi-generator.tech/` (cit. alle pp. 25, 27).

[15] OpenJS Foundation. *Node.js 18 Documentation*. Accessed: 2025-11-03. 2024. URL: `https://nodejs.org/docs/latest-v18.x/api/index.html` (cit. a p. 26).

[16] Microsoft Corporation. *Visual Studio Code Documentation*. Accessed: 2025-11-03. 2024. URL: `https://code.visualstudio.com/docs` (cit. a p. 27).

[17] Stoplight. *Prism: Open-Source API Mocking and Contract Testing*. Accessed: 2025-11-03. 2024. URL: `https://docs.stoplight.io/docs/prism/` (cit. alle pp. 27, 42).

[18] Microsoft Corporation. *TypeScript Documentation*. Accessed: 2025-11-03. 2024. URL: `https://www.typescriptlang.org/docs/` (cit. a p. 30).

[19] TanStack. *TanStack Query v5 Documentation*. Accessed: 2025-11-18. 2024. URL: `https://tanstack.com/query/latest/docs/framework/react/overview` (cit. a p. 30).

[20] OpenJS Foundation. *ESLint: Pluggable JavaScript Linter*. Accessed: 2025-11-03. 2024. URL: `https://eslint.org/docs/latest/` (cit. a p. 40).

[21] James Chester et al. *Prettier: Opinionated Code Formatter*. Accessed: 2025-11-03. 2024. URL: `https://prettier.io/docs/en/index.html` (cit. a p. 40).

[22] Typicode. *Husky: Git hooks made easy*. Accessed: 2025-11-03. 2024. URL: `https://typicode.github.io/husky/` (cit. a p. 40).

[23] Andrey Okunet. *lint-staged Documentation*. Accessed: 2025-11-03. 2024. URL: `https://github.com/lint-staged/lint-staged` (cit. a p. 40).

[24] Meta Platforms. *Jest: Delightful JavaScript Testing*. Accessed: 2025-11-03. 2024. URL: `https://jestjs.io/docs/getting-started` (cit. a p. 45).