Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Sviluppo Backend per la Gestione Immobiliare

Tesi di Laurea

Relatore

Prof. Zanella Marco

Laureando

Zhuo Yi Hao

Matricola 2044939

Anno Accademico 2025-2026

"Quando sviluppi da solo, o ti assumi tutto: backend, frontend, deploy e problemi; oppure non stai davvero costruendo nulla."

— Io stesso.

Ringraziamenti

Desidero esprimere la mia gratitudine al professor Zanella Marco, mio relatore, per l'aiuto e il sostegno che mi ha dato durante la stesura dell'elaborato.

Vorrei anche ringraziare, con affetto, i miei genitori per il loro sostegno, il grande aiuto e la loro presenza in ogni momento durante gli anni di studio.

Desidero poi ringraziare i miei amici per i bellissimi anni trascorsi insieme e le mille avventure vissute.

Padova, 07 2026 *Zhuo Yi Hao*

Sommario

Questo elaborato presenta il lavoro svolto durante il tirocinio curricolare presso la società YE'S GROUP S.R.L., concentrato sulla progettazione e sullo sviluppo del backend di una piattaforma web per la gestione di immobili e prenotazioni. Il progetto è stato realizzato in collaborazione con un altro studente, responsabile del frontend, mentre il mio contributo ha riguardato l'intero livello server-side: analisi dei requisiti, definizione dell'architettura, modellazione dei dati e implementazione dei servizi applicativi.

Nel corso dell'attività ho sviluppato API REST per autenticazione, utenti, proprietà, stanze, prenotazioni, recensioni, prezzi e ruoli, adottando un approccio orientato alla sicurezza e alla manutenibilità. Lo stack tecnologico utilizzato include PHP con Slim, Eloquent ORM, MariaDB, JWT per l'autenticazione, controllo degli accessi basato su permessi, protezioni CSRF, logging strutturato e metriche per il monitoraggio. Sono inoltre stati integrati test automatizzati e documentazione OpenAPI, in modo da supportare sia la qualità del codice sia l'integrazione con il frontend. L'esperienza ha consolidato competenze tecniche su progettazione backend, affidabilità del servizio e gestione operativa di un'applicazione web reale.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici sorgenti

Acronimi e abbreviazioni

API Application Program Interface. 1, 4, 6, 7, 22, 27, 42, 57

CI/CD Continuous Integration/Continuous Deployment. 26, 27, 33, 34, 63, 67

CORS Cross-Origin Resource Sharing. 23, 27, 29, 34, 42, 55, 66

CSRF Cross-Site Request Forgery. 23, 27, 29, 42, 55, 57, 62, 64, 66

HTTP HyperText Transfer Protocol. 22, 23, 26–29, 34–36, 43, 45, 55, 61, 67

JSON JavaScript Object Notation. 24, 27, 28, 46

JWT JSON Web Token. xi, 23, 29, 34–36, 42, 45, 55, 58, 62, 66, 67

ORM Object-Relational Mapping. 23, 24, 42, 43, 61, 62, 66, 67

PDO PHP Data Objects. 23

RBAC Role-Based Access Control. 29, 42, 48, 55, 66, 67

REST Representational State Transfer. 24, 42, 66

UML Unified Modeling Language. 10

VS Code Visual Studio Code. 35

Glossario

boilerplate Codice ripetitivo e poco specifico di dominio, spesso necessario per configurazione o integrazione tecnica, che aumenta il volume di implementazione senza aggiungere logica funzionale distintiva. 24, 27

bus factor Indicatore del rischio legato alla concentrazione della conoscenza nel team: misura quante persone chiave diventano indisponibili prima che il progetto subisca un impatto critico. 33

code-first Approccio di sviluppo in cui i contratti API (ad esempio OpenAPI) vengono derivati dal codice implementato, mantenendo allineati documentazione e comportamento reale del servizio. 23, 32, 39

debito tecnico Il debito tecnico è un concetto che rappresenta il costo implicito di ulteriori lavori causati da una scelta tecnica rapida e facile nel breve termine, invece di utilizzare un approccio migliore che richiederebbe più tempo. Il debito tecnico può accumularsi quando si sacrificano pratiche di sviluppo software di alta qualità per rispettare scadenze o ridurre i costi, portando a problemi di manutenzione, scalabilità e affidabilità del software nel lungo termine. 5

defense in depth Principio di sicurezza che prevede più livelli di controlli indipendenti, così che il fallimento di una singola misura non comprometta l'intero sistema. 29

full-stack Approccio o framework che integra più livelli applicativi (ad esempio routing, business logic, persistenza e tool ausiliari) in un ecosistema unico. 22

git workflow Insieme di regole operative per gestire branching, revisioni, merge e tracciabilità delle modifiche tramite Git. 33, 35

micro-framework Framework minimale focalizzato sulla pipeline HTTP e sull'estensibilità, che offre componenti essenziali lasciando maggiore controllo progettuale al team. 22

middleware Componente software che intercetta una richiesta nella pipeline HTTP per applicare responsabilità trasversali come autenticazione, autorizzazione, logging e metriche. 22

onboarding Processo di inserimento di nuovi membri nel team, con trasferimento di conoscenze, strumenti e convenzioni operative. 23

OpenAPI Standard per descrivere contratti API REST in modo formale e machine-readable, utile per documentazione, validazione e generazione di strumenti client/server. 24, 47, 61, 64, 66, 67

post-mortem Analisi retrospettiva di un incidente o di un malfunzionamento con l'obiettivo di identificarne le cause, valutarne l'impatto e definire azioni correttive e preventive. 25

quality gate Controllo automatico o manuale che una modifica deve superare (test, linting, analisi statica) prima di essere integrata o rilasciata. 26

stakeholder Nel contesto progettuale, gli stakeholder sono i soggetti interessati al sistema e ai suoi risultati, come committenti, team di sviluppo, utenti finali e altre parti coinvolte nelle decisioni di prodotto. 10

trade-off Compromesso tra alternative progettuali in cui il miglioramento di una proprietà (ad esempio controllo o performance) comporta costi o limiti su un'altra. 38

transazione Unità atomica di operazioni sul database che viene completata integralmente o annullata, garantendo coerenza dei dati in presenza di errori o concorrenza. 29

diagramma dei casi d'uso Nel contesto UML, il diagramma dei casi d'uso (Use Case Diagram) rappresenta le funzionalità offerte da un sistema dal punto di vista degli attori che interagiscono con esso. 10

Capitolo 1

Introduzione

Negli ultimi anni il mercato degli affitti ha una significativa trasformazione digitale soprattutto in ambito per gli alloggi universitari. In un contesto urbano come quello padovano, caratterizzato da una domanda abitativa dinamica, emerge la necessità di piattaforme software in grado di coniugare affidabilità operativa, sicurezza dei dati e rapidità evolutiva del prodotto. Fonti istituzionali locali evidenziano inoltre un aumento delle richieste di alloggi a costi sostenibili da parte di studenti e studentesse, in particolare fuori sede Universita degli Studi di Padova. *Coabitazione intergenerazionale, al via la quarta edizione del progetto.* URL: https://www.unipd.it/news/coabitazione-intergenerazionale-al-la-quarta-edizione-del-progetto (visitato il giorno 18/06/2026); Progetto Giovani Padova. *Coabitazione intergenerazionale nel Comune di Padova.* URL: https://www.progettogiovani.pd.it/animazione/coabitazione-intergenerazionale-nel-comune-di-padova/ (visitato il giorno 17/06/2026). In questo scenario si colloca il progetto sviluppato durante lo stage, finalizzato alla realizzazione del backend di una piattaforma web per la gestione di annunci immobiliari e prenotazioni delle stanze.

Il progetto è stato svolto in collaborazione con un altro studente: mentre frontend e pannello amministrativo sono stati sviluppati in parallelo sul lato client, il mio contributo si è concentrato sul livello server-side. Tale suddivisione ha richiesto un coordinamento continuo sulle interfacce tra i due livelli applicativi, in particolare nella progettazione delle *Application Program Inter-*

face (API)$_G$, sulla definizione dei dati trasportati e sulla gestione uniforme degli errori.

L'attività non si è limitata alla sola implementazione di endpoint, ma ha coperto l'intero ciclo di vita del backend: analisi del problema, modellazione concettuale, progettazione architetturale, sviluppo incrementale, test, monitoraggio e documentazione tecnica. L'obiettivo principale è stato costruire un sistema manutenibile e pronto a sostenere l'evoluzione funzionale richiesta da un'applicazione.

1.1 L'azienda

Il tirocinio è stato svolto presso YE'S Group s.r.l., con sede a Padova, attiva nel settore della locazione immobiliare. L'azienda si rivolge con gli affari in particolare a studenti internazionali, con attenzione anche a chi proviene da Paesi extra-UE, offrendo soluzioni abitative pensate per favorire studio, autonomia personale e inserimento nel contesto cittadino.

L'offerta non si limita alla disponibilità degli alloggi, ma include un accompagnamento continuativo durante la permanenza: supporto operativo, gestione delle esigenze quotidiane e mediazione nei passaggi più delicati dell'integrazione. In questa prospettiva, la qualità del servizio dipende sia dall'organizzazione interna sia dalla capacità di mantenere processi chiari, tracciabili e tempestivi.

Dal punto di vista organizzativo, YE'S Group s.r.l. adotta un'impostazione orientata al cliente, con particolare cura per la qualità percepita e per la comunicazione con gli utenti. Il carattere internazionale dell'ambiente costituisce un elemento distintivo, poiché favorisce la costruzione di una comunità inclusiva in cui gli studenti possano sentirsi accolti e supportati.

In tale contesto si inserisce il progetto di tirocinio, finalizzato allo sviluppo del backend di una piattaforma web per la gestione degli affitti, con l'obiettivo di migliorare l'efficienza dei processi e l'esperienza complessiva degli utenti.

1.2 L'idea

Lo scopo dello stage è stato contribuire allo sviluppo del backend di una piattaforma per la gestione degli affitti, con l'obiettivo di supportare in modo affidabile le principali funzionalità del servizio e migliorare l'efficienza dei processi operativi.

La scelta di questo percorso nasce dall'interesse personale verso la progettazione software lato server e dalla volontà di applicare in un contesto reale le competenze maturate durante il percorso universitario. L'esperienza è risultata coerente con i miei obiettivi formativi, perché ha richiesto di tradurre esigenze concrete in soluzioni tecniche sostenibili.

Dal punto di vista didattico e professionale, lo stage ha rappresentato un passaggio importante dall'apprendimento teorico alla responsabilità su un progetto reale, rafforzando capacità di analisi, collaborazione e gestione progressiva delle attività. I dettagli tecnici relativi a progettazione, strumenti e validazione sono approfonditi nei capitoli successivi.

1.3 Organizzazione del testo

Il secondo capitolo approfondisce il contesto operativo dello stage, l'organizzazione del lavoro e il rapporto con l'azienda e il tutor.

Il terzo capitolo presenta l'analisi dei requisiti funzionali e non funzionali, con la relativa classificazione e priorità.

Il quarto capitolo introduce il quadro teorico e metodologico di riferimento e le scelte tecniche adottate.

Il quinto capitolo descrive il lavoro svolto nella fase di progettazione e codifica del backend.

Il sesto capitolo descrive le attività di verifica e validazione, con i principali problemi incontrati e le soluzioni adottate.

Il settimo capitolo sintetizza i risultati raggiunti, le conoscenze acquisite e i possibili sviluppi futuri.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune vengono definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *API* $_{G}$;
- i termini in lingua straniera o facenti parte del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

2.1 Organizzazione dello stage

Il periodo di stage è stato organizzato con una struttura iterativa, orientata alla consegna progressiva di funzionalità verificabili. L'obiettivo non era solo produrre codice, ma definire un flusso di lavoro stabile tra analisi, implementazione, confronto e revisione.

La pianificazione è stata articolata in iterazioni brevi, ciascuna con obiettivi tecnici espliciti e criteri di completamento condivisi. Il lavoro è stato impostato in modo da ridurre i blocchi tra attività dipendenti e da mantenere la continuità tra sviluppo backend e integrazione con il lato client.

Le fasi principali hanno incluso:

- definizione degli obiettivi dell'iterazione e pianificazione dettagliata;
- sviluppo delle componenti previste;
- allineamento intermedio con azienda e tutor;
- verifica tecnica e chiusura dell'iterazione.

Questo modello ha consentito di mantenere sotto controllo tempi, priorità e qualità, limitando l'accumulo di *debito tecnico*$_G$ nelle fasi finali.

2.2 Rapporto con l'azienda

2.2.1 Modalità di collaborazione

La collaborazione con l'azienda è stata continua e orientata al risultato, con verifiche periodiche dello stato di avanzamento. Le attività sono state guidate da priorità di prodotto, vincoli temporali e sostenibilità delle scelte.

2.2.2 Comunicazione e allineamento dei requisiti

L'allineamento dei requisiti è avvenuto tramite confronto frequente con particolare attenzione alle interfacce tra backend e frontend.

Per limitare ambiguità implementative, le modifiche sono state tracciate e condivise prima del rilascio, in modo da ridurre incompatibilità tra componenti. In particolare, è stata mantenuta coerenza sui contratti delle API_G esposte dal backend.

2.3 Rapporto con il tutor aziendale

2.3.1 Riunioni periodiche e feedback

Il rapporto con il tutor aziendale ha avuto un ruolo centrale nella validazione delle scelte tecniche. Essendo un'azienda piccola, il confronto è stato diretto e focalizzato su aspetti operativi e di qualità del prodotto. Le riunioni periodiche hanno rappresentato un momento di verifica e confronto, utile per:

- verificare lo stato di avanzamento rispetto agli obiettivi pianificati;
- discutere criticità tecniche emergenti e possibili alternative;
- definire le priorità dell'iterazione successiva.

Il feedback ricevuto ha migliorato sia la qualità implementativa sia la tracciabilità delle decisioni progettuali.

2.3.2 Supporto tecnico e validazione delle scelte

Le principali scelte tecniche sono state definite attraverso confronto congiunto, con discussioni su architettura, sicurezza applicativa e comportamento delle *API*$_G$. Il tutor ha contribuito soprattutto alla validazione delle opzioni considerate, mentre analisi e proposta delle soluzioni sono state sviluppate in modo condiviso.

La validazione non si è limitata alla correttezza funzionale, ma ha incluso manutenibilità del codice, robustezza degli endpoint, qualità della documentazione tecnica e verificabilità tramite test.

2.4 Metodologia di lavoro

2.4.1 Approccio iterativo e gestione delle priorità

La metodologia adottata è stata iterativa: ogni ciclo di lavoro ha prodotto un'iterazione funzionale testabile, con priorità assegnate in base a impatto utente, dipendenze tecniche e rischio implementativo.

In particolare, il lavoro è stato organizzato in iterazioni brevi, ciascuna composta da pianificazione, implementazione, verifica e revisione con il tutor aziendale. Questo approccio ha consentito di integrare rapidamente i feedback e di ridurre il rischio di regressioni tra un'iterazione e la successiva.

In pratica, sono state affrontate prima le funzionalità fondanti del dominio (autenticazione, utenti, proprietà, stanze), per poi estendere il sistema a prenotazioni, recensioni, prezzi e gestione avanzata dei ruoli.

2.4.2 Strumenti di tracciamento e controllo qualità

Per il controllo qualità sono stati adottati strumenti e pratiche di base:

- revisione periodica del codice sviluppato;
- verifica funzionale delle iterazioni prima del rilascio;
- monitoraggio delle criticità emerse durante l'integrazione.

I dettagli tecnici delle funzioni implementate e degli strumenti utilizzati sono riportati nei capitoli successivi dedicati a progettazione, codifica, verifica e validazione.

2.5 Analisi preventiva dei rischi

Durante la fase iniziale sono stati identificati rischi tecnici e organizzativi rilevanti, con relative possibili soluzioni per mitigarli.

1. Disallineamento tra backend e frontend

Descrizione: codifica asincrona dei contratti API con possibile incompatibilità dei payload.

Probabilità: medio.

Impatto: alto.

Soluzione: adozione di specifiche OpenAPI aggiornate e revisione congiunta delle modifiche agli endpoint.

2. Gestione incompleta delle autorizzazioni

Descrizione: accesso non corretto a risorse sensibili in assenza di controlli puntuali sui permessi.

Probabilità: medio.

Impatto: alto.

Soluzione: integrazione sistematica di JwtMiddleware e PermissionMiddleware su tutte le rotte protette.

3. Regressioni su funzionalità già rilasciate

Descrizione: introduzione di modifiche con effetti collaterali su parte già validati.

Probabilità: medio.

Impatto: medio.

Soluzione: verifica progressiva degli incrementi e controllo sistematico prima dell'integrazione.

4. Limitata osservabilità del comportamento runtime

Descrizione: difficoltà di individuare colli di bottiglia o errori ricorrenti in ambiente di esecuzione.

Probabilità: basso.

Impatto: medio.

Soluzione: definizione di indicatori operativi e raccolta strutturata delle informazioni di esecuzione.

Capitolo 3

Analisi dei requisiti

3.1 Stakeholder e attori

Gli *stakeholder*$_G$ principali si individuano:

- **Azienda (YE'S Group s.r.l.)**: interessata alla gestione efficiente degli immobili e all'espansione del servizio;
- **Ospite**: utenti finali del sistema, interessati alla ricerca di alloggi;
- **Sviluppatori**: responsabili della manutenzione e dell'evoluzione del sistema.

Gli attori principali del sistema sono:

- **Ospite (utente)**: utilizza il sistema per cercare e visualizzare le proprietà disponibili, inviare richieste di affitto e gestire il proprio account.
- **Amministratore**: gestisce il sistema attraverso il pannello amministrativo, occupandosi di utenti, immobili e prenotazioni.

3.2 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I *diagrammi dei casi d'uso*$_G$ sono diagrammi di tipo *Unified Modeling Language (UML)*$_G$ dedicati alla descrizione delle funzioni o servizi offerti da un sistema,

così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.

3.2.1 Casi d'uso dell'amministratore

Le funzionalità principali dell'amministratore sono rappresentate nelle Figure 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 e 3.7.

UC1a: Registrazione admin

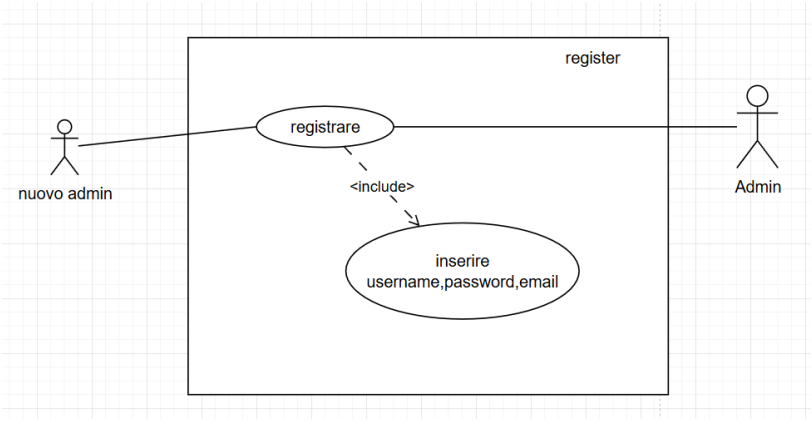

Figura 3.1: Use Case 1: Registrazione admin

Attori Principali: Admin non registrato.

Precondizioni: Un nuovo admin non possiede un account.

Descrizione: L'admin inserisce i propri dati per creare un nuovo account.

Postcondizioni: L'account viene creato e una volta viene autenticato da un admin già registrato, può accedere al sistema.

UC2a: Login al sistema

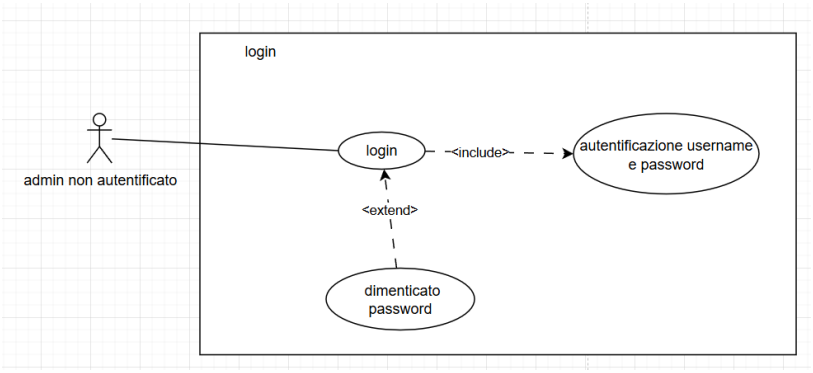

Figura 3.2: Use Case 2: Login al sistema

Attori Principali: Admin non autenticato

Precondizioni: L'utente si trova nella pagina di accesso.

Descrizione: L'utente inserisce username e password per accedere al sistema.

Postcondizioni: L'utente accede al sistema e visualizza il pannello amministrativo.

Scenario Alternativo: Se le credenziali sono errate, il sistema mostra un messaggio di errore.

UC3a: Gestione proprietà

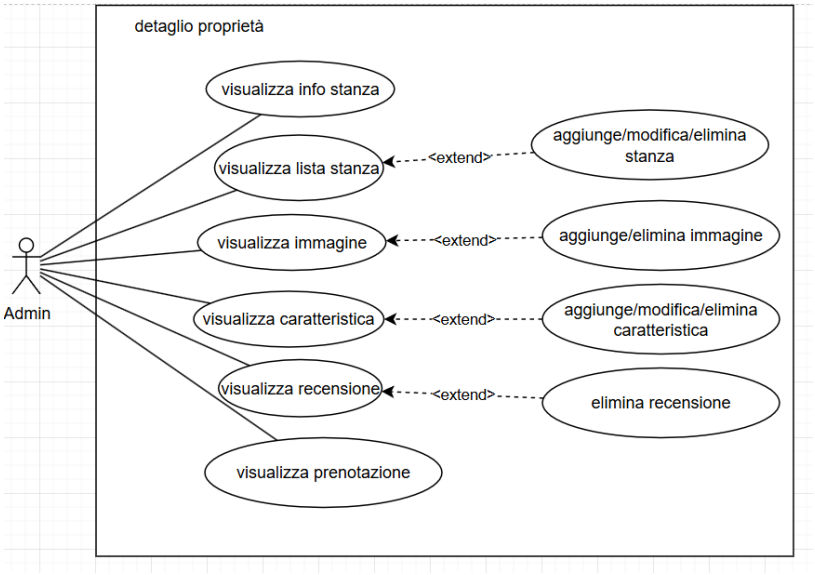

Figura 3.3: Use Case 3: Gestione proprietà

Attori Principali: Amministratore.

Precondizioni: L'amministratore ha effettuato l'accesso al sistema.

Descrizione: L'amministratore può aggiungere, modificare o eliminare una proprietà.

Postcondizioni: Le modifiche alle proprietà vengono salvate nel sistema.

UC4a: Gestione stanze

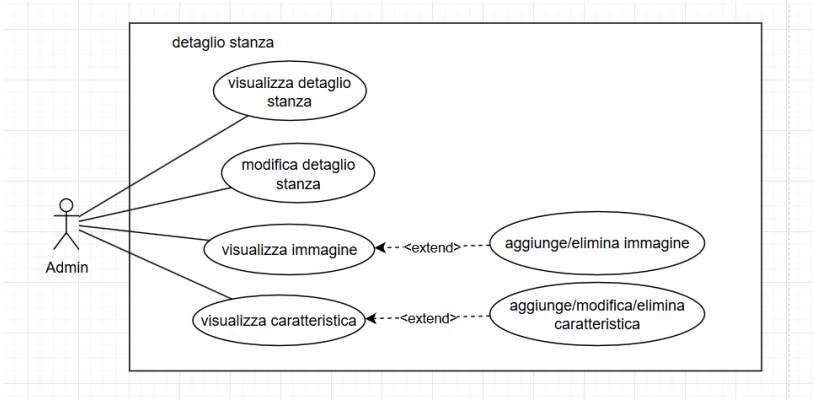

Figura 3.4: Use Case 4: Gestione stanze

Attori Principali: Amministratore.

Precondizioni: L'amministratore ha selezionato una proprietà.

Descrizione: L'amministratore può aggiungere, modificare o eliminare le stanze associate a una proprietà.

Postcondizioni: Le modifiche alle stanze vengono salvate correttamente.

UC5a: Gestione prenotazioni

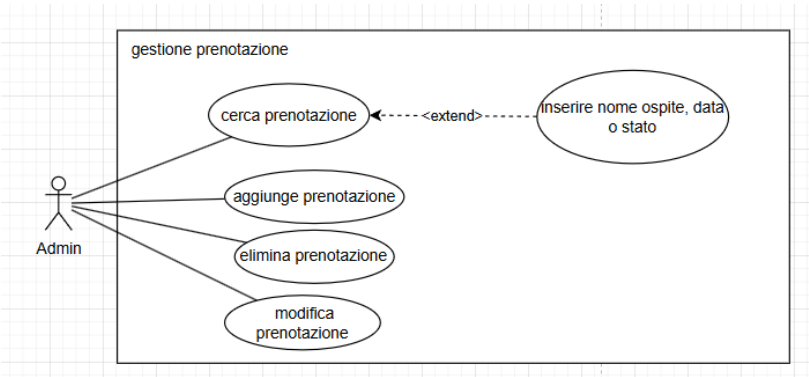

Figura 3.5: Use Case 5: Gestione prenotazioni

Attori Principali: Amministratore.

Precondizioni: L'amministratore ha effettuato l'accesso al sistema.

Descrizione: L'amministratore può visualizzare, aggiungere, modificare o eliminare le prenotazioni.

Postcondizioni: Le informazioni delle prenotazioni vengono aggiornate nel sistema.

UC6a: Gestione utenti

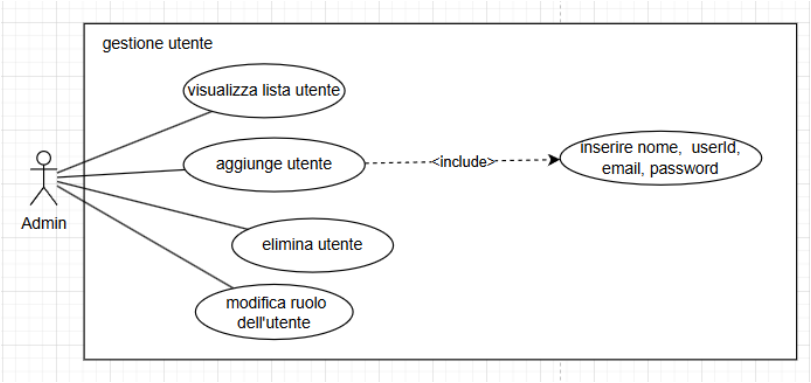

Figura 3.6: Use Case 6: Gestione utenti

Attori Principali: Amministratore.

Precondizioni: L'amministratore ha effettuato l'accesso al sistema.

Descrizione: L'amministratore può visualizzare, aggiungere, eliminare utenti e gestire i loro permessi.

Postcondizioni: Le modifiche agli utenti vengono salvate nel sistema.

UC7a: Gestione profilo

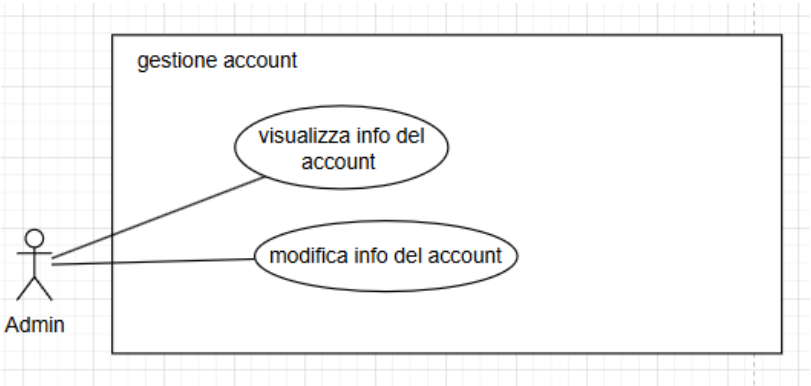

Figura 3.7: Use Case 7: Gestione profilo

Attori Principali: Amministratore.

Precondizioni: L'amministratore è autenticato nel sistema.

Descrizione: L'amministratore può visualizzare e modificare le informazioni del proprio account.

Postcondizioni: Le modifiche vengono salvate correttamente.

3.2.2 Casi d'uso dell'ospite

I casi d'uso dell'ospite sono sintetizzati nelle Figure 3.8, 3.9 e 3.10.

UC1o: Registrazione utente

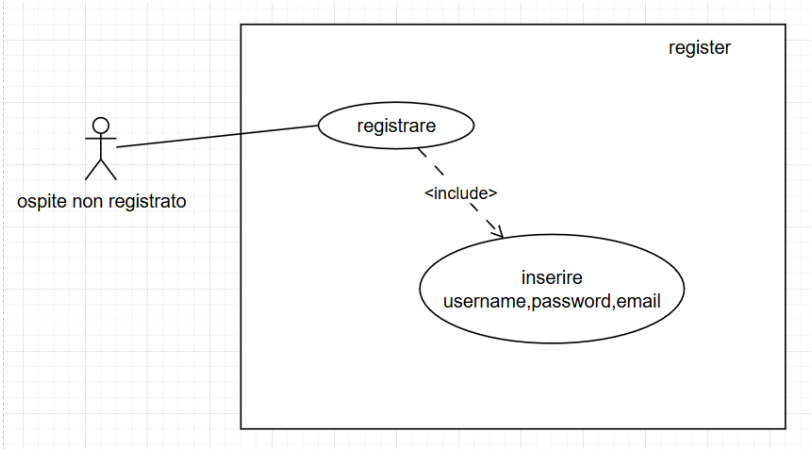

Figura 3.8: Use Case 1: Registrazione utente

Attori Principali: Ospite non registrato.

Precondizioni: Un nuovo ospite non possiede un account.

Descrizione: L'ospite inserisce i propri dati per creare un nuovo account.

Postcondizioni: L'account viene creato e l'utente può accedere al sito.

UC2o: Login al sito

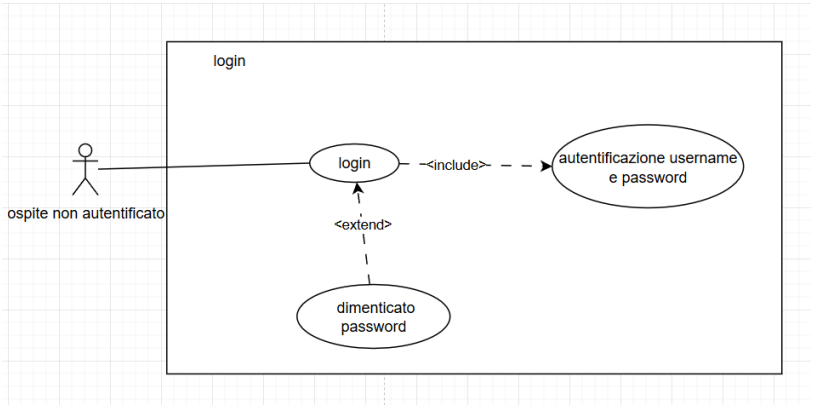

Figura 3.9: Use Case 2: Login al sito

Attori Principali: Ospite non autenticato.

Precondizioni: L'utente si trova nella pagina di accesso.

Descrizione: L'utente inserisce username e password per accedere al sito.

Postcondizioni: L'utente accede al sito e può visualizzare le proprietà disponibili.

Scenario Alternativo: Se le credenziali sono errate, il sistema mostra un messaggio di errore.

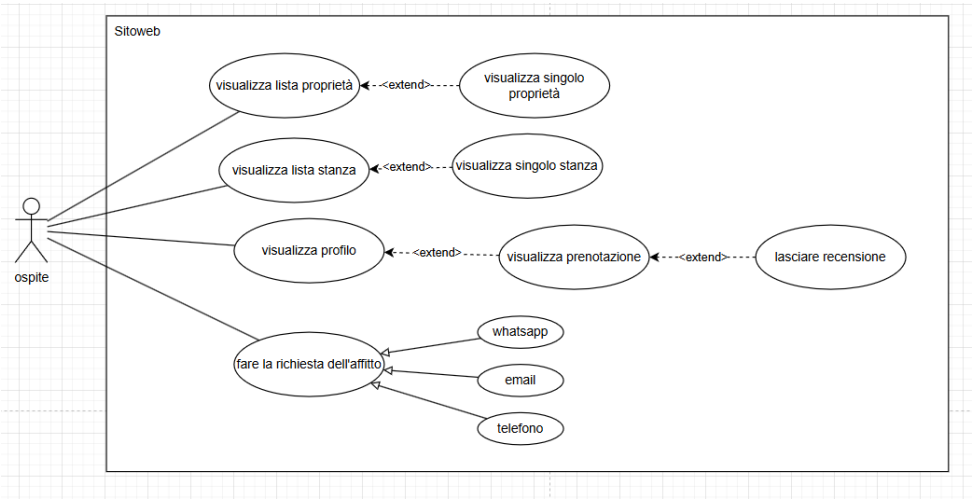

Figura 3.10: Use Case 3,4,5,6: Attività principali dell'ospite

UC3o: Visualizzazione lista proprietà

Attori Principali: Ospite non autenticato, Ospite autenticato.

Precondizioni: L'utente accede al sito web.

Descrizione: L'utente visualizza l'elenco delle proprietà disponibili.

Postcondizioni: Viene mostrata la lista delle proprietà.

UC4o: Visualizzazione dettaglio proprietà

Attori Principali: Ospite non autenticato, Ospite autenticato.

Precondizioni: L'utente ha selezionato una proprietà dalla lista.

Descrizione: L'utente visualizza i dettagli della proprietà, comprese stanze, immagini e caratteristiche.

Postcondizioni: Vengono mostrati i dettagli completi della proprietà.

UC5o: Gestione profilo

Attori Principali: Ospite autenticato.

Precondizioni: L'utente è autenticato nel sito.

Descrizione: L'utente può visualizzare e modificare le informazioni del proprio account.

Postcondizioni: Le modifiche vengono salvate correttamente.

UC6o: Richiesta di affitto

Attori Principali: Ospite autenticato.

Precondizioni: L'utente ha selezionato una proprietà o una stanza.

Descrizione: L'utente invia una richiesta di affitto tramite uno dei canali disponibili (email, telefono o WhatsApp).

Postcondizioni: La richiesta viene inviata correttamente.

3.3 Tracciamento dei requisiti

Si farà riferimento ai requisiti secondo le seguenti notazioni:

- **O**: per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
- **D**: per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
- **F**: per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.

Le lettere precedentemente indicate saranno seguite da una coppia sequenziale di numeri, identificativo del requisito. Si prevede lo svolgimento dei seguenti obiettivi: La mappatura completa tra requisiti e casi d'uso è riportata nella Tabella 3.1.

Requisiti obbligatori (O):

- O01: progettare e implementare un database MariaDB normalizzato;
- O02: creare API RESTful sicure in PHP;
- O03: implementare autenticazione e autorizzazione utenti;
- O04: integrare logging, metriche e monitoraggio backend;
- O05: automatizzare test e deploy tramite CI/CD.

Requisiti desiderabili (D):

- D01: migliorare le prestazioni tramite caching (Redis o similare);
- D02: analizzare metriche di utilizzo e tempi di risposta delle API;
- D03: documentare comparativamente i benefici e i miglioramenti ottenuti.

Requisiti facoltativi (F):

- F01: estendere le API per supporto multi-tenant;
- F02: integrare sicurezza avanzata (rate limiting, anomaly detection).

La pianificazione della parte backend del progetto è stata definita al fine di organizzare lo sviluppo in modo strutturato e progressivo. Il piano di lavoro è stato suddiviso in attività settimanali, con l'obiettivo di completare il progetto progressivamente, mantenendo un ricco feedback e confronto con i membri del team e con il tutor aziendale.

Tabella 3.1: Tracciamento dei requisiti backend e casi d'uso associati

Requisito	**Descrizione**	**Use Case**
O01	Progettazione e implementazione database MariaDB normalizzato	UC3a, UC4a, UC5a, UC6a, UC3o, UC4o
O02	Creazione di API RESTful sicure in PHP	UC3a, UC4a, UC5a, UC6a, UC3o, UC4o, UC6o
O03	Implementazione autenticazione e autorizzazione utenti	UC1a, UC2a, UC6a, UC1o, UC2o, UC5o
O04	Integrazione logging, metriche e monitoraggio backend	-
O05	Automazione test e deploy tramite CI/CD	-
D01	Miglioramento prestazioni tramite caching (Redis o similare)	UC3o, UC4o, UC5a
D02	Analisi metriche di utilizzo e tempi di risposta API	-
D03	Documentazione comparativa dei miglioramenti ottenuti	-
F01	Estensione API per supporto multi-tenant	-
F02	Integrazione sicurezza avanzata (rate limiting, anomaly detection)	-

3.4 Pianificazione

La pianificazione della parte backend del progetto è stata definita al fine di organizzare lo sviluppo in modo strutturato e progressivo. Il piano di lavoro è stato suddiviso in attività settimanali, con l'obiettivo di completare il progetto progressivamente, mantenendo un ricco feedback e confronto con i membri del team e con il tutor aziendale. La pianificazione operativa è riassunta nella Tabella 3.2.

Settimana	Ore	Attività principali
1	40	O01: Analisi requisiti, progettazione database e schema ER
2	40	O02: Setup ambiente PHP e sviluppo API base
3	40	O03: Implementazione autenticazione JWT e ruoli utenti
4	40	O04: Integrazione logging e monitoraggio (Prometheus)
5	40	O05: Testing API e ottimizzazione query
6	40	D01: Caching e ottimizzazione prestazioni
7	40	D02: CI/CD pipeline e documentazione API
8	40	D03, F01: Report risultati ed estensione API multi-tenant

Tabella 3.2: Pianificazione settimanale delle attività backend

Capitolo 4

Introduzione teorica

Obiettivo del capitolo: presentare il quadro teorico, i criteri metodologici e le scelte tecnologiche che orientano lo sviluppo del backend YesHomeGroup, mantenendo coerenza tra architettura, sicurezza, qualità e verificabilità del sistema.

4.1 Tecnologie esistenti nel dominio applicativo

4.1.1 Alternative per lo sviluppo backend

Nel dominio delle applicazioni web dedicate alla gestione di prenotazioni e contenuti immobiliari, si riscontrano tre famiglie di soluzioni backend: framework *full-stack*$_G$, *micro-framework*$_G$ e piattaforme server-side basate su runtime JavaScript.

I framework full-stack (ad esempio Laravel) mettono a disposizione un ecosistema esteso e consentono un avvio rapido del progetto; Tuttavia, rischiano di essere troppo pesanti o rigidi se devi solo costruire delle semplici *API*$_G$.

I micro-framework (come Slim), al contrario, privilegiano leggerezza e controllo esplicito della pipeline *HyperText Transfer Protocol (HTTP)*$_G$, risultando particolarmente adeguati quando si intende mantenere una separazione netta tra routing, *middleware*$_G$, controller e logica di dominio.

Le soluzioni Node.js/TypeScript presentano vantaggi in termini di uniformità tecnologica con il frontend, ma non rappresentavano il focus formativo del

presente tirocinio, orientato all'ecosistema PHP.

Alla luce di tali considerazioni, nel progetto YesHomeGroup è stato adottato un approccio *code-first*$_G$ orientato alle API basato su Slim 4, ritenuto coerente con gli obiettivi di modularità, tracciabilità delle richieste e integrazione con il client web. Le proprietà teoriche di questo approccio sono coerenti con la documentazione ufficiale di Slim e con gli standard PSR-7/PSR-15 *Slim Framework 4 Documentation*. URL: https://www.slimframework.com/docs/v4/ (visitato il giorno 17/06/2026); *PSR-7: HTTP Message Interfaces*. URL: https://www.php-fig.org/psr/psr-7/ (visitato il giorno 17/06/2026); *PSR-15: HTTP Server Request Handlers*. URL: https://www.php-fig.org/psr/psr-15/ (visitato il giorno 17/06/2026).

Nella valutazione iniziale sono stati considerati i seguenti aspetti:

- controllo della pipeline *HTTP*$_G$ e facilità di estensione tramite middleware custom (ad es. inserimento esplicito di middleware *JWT*$_G$, *Cross-Site Request Forgery (CSRF)*$_G$ e logging);
- allineamento con i requisiti di sicurezza e monitoraggio (ad es. gestione *Cross-Origin Resource Sharing (CORS)*$_G$, metriche Prometheus ed error tracking applicativo);
- sostenibilità operativa: curva di apprendimento per il team e costo di manutenzione nel medio periodo (ad es. convenzioni condivise e minore complessità di *onboarding*$_G$).

L'approccio micro-framework è risultato più appropriato soprattutto perché rende espliciti i punti di attraversamento della richiesta (autenticazione, autorizzazione, validazione CSRF, logging e metriche), evitando livelli impliciti che tendono a complicare analisi, debug e attività di test.

4.1.2 Alternative per persistenza e integrazione dati

Per il livello di persistenza sono state considerate tre opzioni: accesso SQL diretto (*PHP Data Objects (PDO)*$_G$), query builder e *Object-Relational Mapping*

(ORM)$_G$. L'approccio SQL puro garantisce il massimo controllo, ma richiede una quantità maggiore di codice; il query builder riduce parte del *boilerplate*$_G$, pur mantenendo una gestione manuale non trascurabile delle relazioni.

L'adozione di Eloquent (*ORM*$_G$) ha permesso una modellazione più espressiva delle entità applicative, incluse le relazioni tra utenti, ruoli, permessi, proprietà, stanze, prenotazioni e recensioni. In ambiente di produzione la persistenza è affidata a MariaDB, mentre in test si utilizza SQLite in memoria, così da ottenere esecuzioni rapide e isolate. Questa scelta è supportata dalle linee guida ufficiali di Eloquent e dalla documentazione tecnica di MariaDB *Eloquent: Getting Started*. URL: https://laravel.com/docs/11.x/eloquent (visitato il giorno 17/06/2026); *MariaDB Documentation*. URL: https://mariadb.com/docs/ (visitato il giorno 18/06/2026).

Per l'integrazione frontend-backend è stato adottato lo stile *Representational State Transfer (REST)*$_G$ con payload *JavaScript Object Notation (JSON)*$_G$, distinguendo endpoint pubblici e protetti e formalizzando i contratti tramite specifiche *OpenAPI*$_G$ generate dal codice. La preparazione teorica su ingegneria del software e progettazione orientata a componenti riusabili è stata supportata anche da materiale didattico e testi di riferimento del corso Universita degli Studi di Padova. *Materiale didattico del corso*. Il materiale didattico presentato durante le lezioni e stato pubblicato progressivamente in formato digitale, collegato alle singole lezioni, congiuntamente alla registrazione della medesima lezione dell'anno accademico 2020/2021. 2021; Ian Sommerville. *Software Engineering*. 10ª ed. Pearson Education e Addison-Wesley, 2015; Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. La formalizzazione dei contratti segue lo standard OpenAPI e l'uso di swagger-php nel flusso code-first *OpenAPI Specification*. URL: https://spec.openapis.org/oas/latest.html (visitato il giorno 17/06/2026); *swagger-php*. URL: https://github.com/zircote/swagger-php (visitato il giorno 18/06/2026).

Dal punto di vista dei pattern di integrazione dati, le scelte principali sono state le seguenti:

- **modello relazionale**: adeguato alla natura strutturata del dominio (utenti, ruoli, prenotazioni, listini, recensioni);
- **relazioni esplicite**: utili a evitare duplicazioni e inconsistenze;
- **coerenza sulle prenotazioni**: validazioni di sovrapposizione e controlli applicativi per prevenire conflitti temporali;
- **bootstrap differenziato per test**: supporto SQLite in-memory per ridurre tempi di feedback.

4.1.3 Alternative per osservabilità e monitoraggio

Per l'osservabilità sono stati analizzati tre livelli progressivi:

- solo logging applicativo;
- logging + metriche aggregate;
- logging + metriche + alerting strutturato.

Il solo logging risulta utile nelle analisi *post-mortem*$_G$, ma meno efficace nell'individuazione tempestiva di regressioni. L'integrazione di metriche Prometheus (volume richieste, latenza, errori, login riusciti/falliti, eventi applicativi) consente invece monitoraggio continuo e costruzione di dashboard orientate al comportamento del servizio. L'impostazione segue il modello osservazionale descritto dalla documentazione di Prometheus e dalle pratiche di logging con Monolog *Prometheus Overview*. URL: https://prometheus.io/docs/introduction/overview/ (visitato il giorno 17/06/2026); *Monolog - Logging for PHP*. URL: https://github.com/Seldaek/monolog (visitato il giorno 18/06/2026).

4.1.4 Alternative per testing e quality gate

Anche la strategia di qualità è stata oggetto di valutazione:

- test manuale esclusivo (rapido ma fragile);

- test unitari isolati (buon controllo locale, copertura parziale dei flussi *HTTP*$_G$ reali);
- combinazione unit + integration + *quality gate*$_G$ statici.

La soluzione adottata corrisponde alla terza opzione: PHPUnit per test unitari e di integrazione, affiancato da strumenti di analisi statica e conformità stilistica (PHP_CodeSniffer, PHP-CS-Fixer, PHPStan). Nella pipeline *Continuous Integration/Continuous Deployment (CI/CD)*$_G$ attuale sono eseguiti in modo sistematico linting (PHPCS) e test (PHPUnit), mentre gli altri quality gate sono integrati nel workflow di sviluppo. I riferimenti tecnici principali per questi strumenti sono la documentazione ufficiale di PHPUnit, PHPCS, PHP-CS-Fixer, PHPStan e GitHub Actions *PHPUnit Manual*. URL: https://docs.phpunit.de/en/11.5/ (visitato il giorno 17/06/2026); PHPCSStandards. *PHP_CodeSniffer*. URL: https://github.com/PHPCSStandards/PHP_CodeSniffer (visitato il giorno 18/06/2026); *PHP Coding Standards Fixer*. URL: https://github.com/PHP-CS-Fixer/PHP-CS-Fixer (visitato il giorno 18/06/2026); *PHPStan - PHP Static Analysis Tool*. URL: https://github.com/phpstan/phpstan (visitato il giorno 18/06/2026); *GitHub Actions Documentation*. URL: https://docs.github.com/en/actions (visitato il giorno 18/06/2026).

4.1.5 Confronto sintetico delle opzioni considerate

Area	**Opzione**	**Vantaggi**	**Limiti**
Backend framework	Framework completo (full-stack)	Avvio rapido e molti strumenti pronti	Più vincoli architetturali e meno controllo fine della pipeline
Backend framework	Framework leggero (micro-framework)	Struttura essenziale, maggiore controllo e modularità	Richiede più progettazione iniziale

Persistenza	SQL diretto	Controllo totale sulle query	Più *boilerplate*$_G$ e maggiore rischio di errori ripetitivi
Persistenza	ORM	Modellazione più rapida di entità e relazioni	Richiede attenzione su query complesse e prestazioni
Osservabilità	Solo log	Facile da introdurre	Supporto limitato a trend e alert operativi
Osservabilità	Log + metriche	Diagnosi più rapida e monitoraggio continuo	Richiede convenzioni chiare su nomi e label metriche
Testing	Solo manuale	Nessun setup iniziale complesso	Regressioni più difficili da intercettare in modo sistematico
Testing	Unit + Integration + *CI/CD*$_G$	Copertura multilivello e feedback più rapido	Maggiore investimento iniziale sulla test suite

4.2 Aspetti teorici di riferimento

4.2.1 Progettazione di API e separazione dei livelli

La progettazione delle *API*$_G$ segue principi resource-oriented: URI stabili, uso coerente dei metodi *HTTP*$_G$, codici di stato semanticamente significativi e rappresentazioni *JSON*$_G$ omogenee. Tale impostazione rende il sistema prevedibile per i client e favorisce manutenzione e verificabilità.

Dal punto di vista architetturale, il flusso applicativo è organizzato in livelli:

- **Routes**: definiscono endpoint e composizione dei middleware;
- **Middleware**: gestiscono cross-cutting concerns (autenticazione, autorizzazione, *CORS*$_G$, *CSRF*$_G$, logging, metriche);

- **Controllers**: coordinano validazione input e output *HTTP*$_G$;
- **Services/Models**: incapsulano logica applicativa e accesso ai dati.

Questa separazione riduce l'accoppiamento tra componenti, migliora la testabilità e consente evoluzioni incrementali limitando gli impatti trasversali.

Oltre alla separazione tecnica, sono stati adottati anche principi operativi:

- endpoint orientati a risorse e non a procedure generiche;
- naming coerente (plurali per collezioni, identificatori in path);
- payload prevedibili, con errori *JSON*$_G$ standardizzati;
- responsabilità univoca per ogni middleware.

4.2.2 Semantica HTTP, idempotenza e gestione errori

La semantica *HTTP*$_G$ è stata applicata in modo esplicito:

- **GET**: sola lettura, senza effetti collaterali;
- **POST**: creazione o operazioni non idempotenti;
- **PUT**: aggiornamento idempotente di risorsa nota;
- **DELETE**: rimozione, idealmente idempotente lato client.

La classificazione degli errori è stata organizzata per categorie:

- **4xx**: input non valido, autenticazione/permessi assenti, conflitti applicativi;
- **5xx**: errori inattesi lato servizio o dipendenze.

Questa impostazione semplifica l'integrazione con il frontend, poiché permette di distinguere in modo netto errori recuperabili e condizioni non recuperabili.

4.2.3 Sicurezza applicativa e controllo degli accessi

Il modello di sicurezza implementato combina meccanismi complementari. L'autenticazione è basata su *JWT*$_G$ (Bearer token), mentre l'autorizzazione adotta un modello *Role-Based Access Control (RBAC)*$_G$ con permessi granulari (ad esempio *property:create*, *booking:admin_edit*).

Per mitigare richieste cross-site non autorizzate, le operazioni mutanti (POST/PUT/DELETE) richiedono token *CSRF*$_G$. La policy *CORS*$_G$ è configurabile tramite variabili d'ambiente per controllare origini, metodi e header ammessi.

La sicurezza è ulteriormente rafforzata da validazioni lato server, logging strutturato degli eventi e raccolta di metriche su tentativi di autenticazione, errori *HTTP*$_G$ e anomalie operative.

In termini di *defense in depth*$_G$, le contromisure sono distribuite su livelli distinti:

- **confine *HTTP*$_G$**: *CORS*$_G$ e preflight;
- **identità**: *JWT*$_G$ con scadenza e verifica firma;
- **autorizzazione**: controllo per permesso su singola rotta;
- **integrità richiesta**: token CSRF su metodi mutanti;
- **osservabilità sicurezza**: metriche su login falliti e risposte d'errore.

4.2.4 Coerenza dei dati e gestione transazioni

Il dominio delle prenotazioni richiede particolare attenzione ai temi di coerenza e concorrenza. In presenza di operazioni concorrenti sul medesimo intervallo temporale, la logica applicativa deve prevenire sovrapposizioni non valide.

Nel backend corrente, la coerenza del prenotazioni è garantita principalmente da controlli applicativi di disponibilità (verifica di overlap) prima della persistenza e dalla gestione esplicita dei conflitti. L'uso di *transazioni*$_G$ è adottato nei

servizi che richiedono atomicità; in scenari ad alta concorrenza, l'estensione con meccanismi transazionali o locking sui flussi di booking costituisce un'evoluzione naturale. A livello progettuale, ciò si traduce in:

- validazioni di disponibilità prima della persistenza;
- monitoraggio dei conflitti tramite metriche dedicate;
- gestione esplicita dei conflitti con risposte verso il client.

4.2.5 Osservabilità: metriche, log e feedback loop

La raccolta congiunta di log e metriche abilita un ciclo di miglioramento continuo:

1. rilevazione di deviazioni (aumento latenza/errori);
2. diagnosi rapida attraverso correlazione endpoint-log;
3. intervento (ottimizzazione query, tuning middleware, fix funzionale);
4. verifica dell'effetto tramite serie temporali.

Questo approccio contribuisce a ridurre il tempo medio di individuazione e risoluzione dei problemi, migliorando affidabilità percepita e controllo del servizio.

4.2.6 Vista logica dell'architettura adottata

La Figura 4.1 sintetizza la catena logica di elaborazione dal client fino alla persistenza e ai componenti di osservabilità.

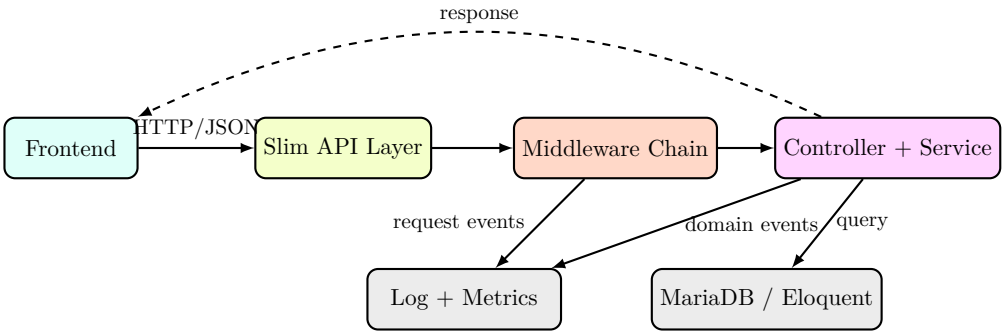

Figura 4.1: Vista logica della catena di elaborazione backend

In questa architettura, il frontend comunica con il backend tramite richieste HTTP/JSON, che vengono gestite da Slim. La catena di middleware applica controlli di sicurezza, logging e raccolta metriche prima di raggiungere i controller, che coordinano la logica applicativa e l'accesso ai dati tramite Eloquent. Le metriche e i log raccolti durante l'elaborazione supportano il monitoraggio continuo e la diagnosi operativa.

4.2.7 Flusso di una richiesta protetta

Il dettaglio dei controlli applicati su una rotta protetta è mostrato nella Figura 4.2.

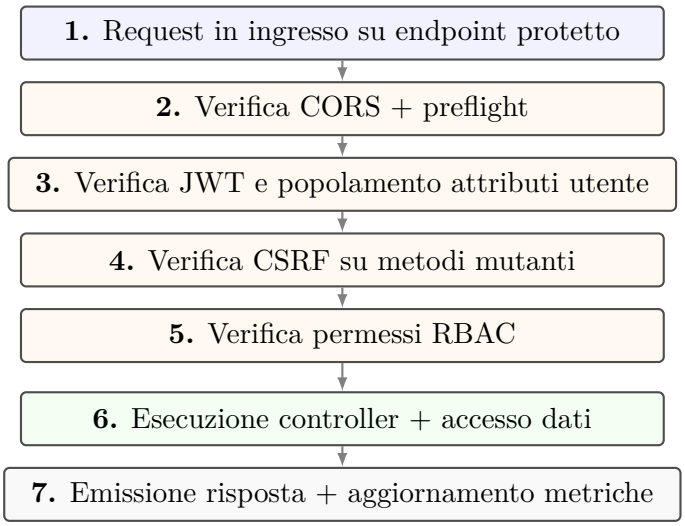

Figura 4.2: Pipeline concettuale di una richiesta autenticata e autorizzata

In questo flusso, una richiesta che raggiunge un endpoint protetto attraversa una serie di middleware che applicano controlli di sicurezza (CORS, JWT,

CSRF, RBAC) prima di raggiungere il controller responsabile dell'elaborazione della logica applicativa. Al termine dell'esecuzione del controller, viene emessa la risposta e aggiornate le metriche operative.

4.3 Criteri di scelta degli strumenti

4.3.1 Criteri tecnici

La selezione degli strumenti è stata guidata da criteri espliciti e verificabili:

- **Aderenza allo stile *code-first*$_G$**: supporto a routing esplicito, middleware componibili e contratti OpenAPI derivati dal codice;
- **Manutenibilità**: chiarezza della struttura del codice e separazione delle responsabilità;
- **Osservabilità**: disponibilità di logging applicativo e metriche Prometheus;
- **Testabilità**: supporto a test unitari e di integrazione con bootstrap isolato;
- **Qualità del codice**: integrazione con strumenti automatici di linting, formatting e analisi statica.

Per ridurre l'ambiguità decisionale, tali criteri sono stati applicati in modo sistematico nel confronto tra alternative, privilegiando soluzioni capaci di migliorare simultaneamente leggibilità, robustezza e rapidità del feedback.

4.3.2 Criteri organizzativi e di manutenzione

Accanto ai criteri tecnici, sono stati considerati fattori organizzativi: facilità di onboarding, coerenza con le competenze previste dal tirocinio, chiarezza documentale e ripetibilità dei processi di build/test.

L'adozione di convenzioni PSR-12, pipeline *CI/CD*$_G$ e test automatizzati ha ridotto il rischio di regressioni, semplificando la collaborazione tra componenti tecniche e interlocutori di progetto.

Ulteriori aspetti organizzativi rilevanti:

- riduzione del *bus factor*$_G$ tramite convenzioni esplicite;
- allineamento tra codice e documentazione API;
- auditabilità delle modifiche tramite *git workflow*$_G$ e controlli automatici;
- tempi di revisione contenuti grazie a standard condivisi.

4.3.3 Matrice decisionale sintetica

Criterio	Peso	Priorità	Impatto sulla scelta
Controllo pipeline *HTTP*$_G$	Alto	5/5	Favorisce Slim con middleware dedicati
Sicurezza applicativa	Alto	5/5	Favorisce sicurezza multilivello (*JWT*$_G$, RBAC, CSRF, *CORS*$_G$)
Produttività dati	Medio-Alto	4/5	Favorisce Eloquent rispetto a SQL diretto
Osservabilità operativa	Medio-Alto	4/5	Favorisce monitoraggio con Prometheus e Monolog
Qualità e mantenibilità	Alto	5/5	Favorisce *CI/CD*$_G$ con test e linting automatici
Costo onboarding	Medio	3/5	Favorisce strumenti già noti al team PHP

Tabella 4.2: Sintesi dei criteri utilizzati per la selezione tecnologica

La Tabella 4.2 rende esplicito il peso attribuito ai criteri di selezione e il relativo impatto sulle scelte finali.

4.4 Strumenti selezionati e motivazioni

Gli strumenti adottati nel backend YesHomeGroup risultano coerenti con i criteri delineati:

- **Slim 4 (PSR-7/PSR-15)**: micro-framework leggero per API *HTTP*$_G$ con pipeline middleware chiara;
- **Eloquent ORM + MariaDB**: modellazione dati espressiva e gestione delle relazioni con buona produttività;
- **firebase/php-jwt**: gestione token *JWT*$_G$ per autenticazione stateless;
- **RBAC custom**: permessi granulari applicati tramite middleware dedicato;
- **Monolog**: logging strutturato su file rotanti per diagnosi e auditing;
- **promphp/prometheus_client_php**: raccolta metriche applicative con endpoint `/metrics`;
- **swagger-php + Swagger UI**: documentazione OpenAPI allineata al codice;
- **PHPUnit + Mockery**: copertura di unit test e integration test su controller, middleware e bootstrap;
- **PHP_CodeSniffer, PHP-CS-Fixer, PHPStan**: controllo continuo della qualità del codice;
- ***Visual Studio Code (VS Code)***$_G$: ambiente di sviluppo principale per implementazione, debug e revisione del codice;
- **GitHub Actions**: esecuzione automatica di lint (PHPCS) e test (PHPUnit) a ogni push.

GitHub

GitHub è stato utilizzato come piattaforma di versionamento del codice e di collaborazione tra i membri del team, a supporto del *git workflow*$_G$ e della tracciabilità delle modifiche.

Strumenti di comunicazione

WeChat e WhatsApp sono stati utilizzati per la comunicazione operativa tra i membri del team e con il tutor aziendale.

La combinazione di tali strumenti ha consentito di bilanciare rapidità di sviluppo, qualità del prodotto e controllo operativo.

4.4.1 Layer HTTP e organizzazione del codice

Slim è stato scelto perché consente di modellare in modo chiaro la catena richiesta-risposta. L'uso esteso dei middleware rende espliciti i punti di controllo e facilita la verificabilità del comportamento applicativo.

4.4.2 Sicurezza: autenticazione, autorizzazione e integrità richiesta

La protezione delle rotte avviene per composizione:

- *JWT*$_G$ per identificare l'utente autenticato;
- middleware permessi per validare i privilegi sul singolo endpoint;
- CSRF middleware per bloccare richieste mutanti prive di token valido.

Da un punto di vista teorico, questo schema riduce l'accoppiamento tra logica di business e logica di sicurezza, poiché ogni controllo viene applicato in modo dichiarativo al livello appropriato della pipeline *HTTP*$_G$.

Le principali proprietà desiderate del modello sono:

- **separazione dei compiti**: autenticazione e autorizzazione non dipendono dai singoli controller;
- **riusabilità**: lo stesso controllo può essere applicato a gruppi di endpoint diversi;

- **tracciabilità**: è possibile analizzare con precisione in quale fase una richiesta viene rifiutata.

4.4.3 Osservabilità e monitoraggio

L'osservabilità viene implementata attraverso metriche applicative e logging. Le metriche coprono richieste totali, latenza, errori e indicatori di dominio (ad esempio esiti di autenticazione e operazioni su prenotazioni).

Sul piano teorico, una buona strategia di osservabilità dovrebbe includere almeno:

- metriche di traffico (volume richieste, distribuzione per endpoint);
- metriche di performance (latenza media e percentili);
- metriche di affidabilità (errori 4xx/5xx, tassi di fallimento);
- eventi applicativi ad alto valore diagnostico (autenticazioni fallite, conflitti operativi).

L'obiettivo non è la sola raccolta di dati, bensì il supporto alle decisioni tecniche: capacity planning, individuazione dei colli di bottiglia, prioritizzazione dei refactoring e verifica dell'efficacia degli interventi correttivi.

4.4.4 Documentazione OpenAPI e allineamento con il codice

La documentazione API viene mantenuta in prossimità del codice applicativo tramite attributi OpenAPI nei controller. Questo approccio riduce la divergenza tra implementazione reale e documentazione esposta ai consumer.

In generale, la documentazione tecnica è efficace quando possiede tre proprietà:

- **coerenza**: i contratti documentati riflettono il comportamento reale del servizio;

- **aggiornabilità**: il costo di manutenzione rimane contenuto;
- **fruibilità**: il consumer riesce a comprendere rapidamente endpoint, payload e vincoli.

4.4.5 Qualità del codice e automazione

La strategia di quality assurance combina linting e test automatici in pipeline, con formattazione e analisi statica integrate nel workflow di sviluppo.

In prospettiva teorica, i quality gate automatizzati forniscono benefici rilevanti:

- riduzione delle regressioni introdotte da modifiche locali;
- uniformità stilistica e semantica del codice;
- feedback rapido per il team durante le iterazioni;
- maggiore affidabilità del processo di integrazione continua.

4.4.6 Trade-off della soluzione tecnologica

Ogni scelta comporta benefici e costi. Nella configurazione adottata, i principali *trade-off* $_G$ teorici sono:

- micro-framework: più controllo ma maggiore responsabilità progettuale;
- ORM: alta produttività ma necessità di particolare attenzione sulle query complesse;
- osservabilità avanzata: diagnosi migliore ma overhead operativo da governare;
- pipeline quality gate: qualità superiore ma investimento iniziale più alto.

4.5 Processo sviluppo prodotto

Nel presente capitolo, il processo di sviluppo viene descritto in termini metodologici generali, senza entrare nel dettaglio delle singole implementazioni.

4.5.1 Modelli di processo considerati

I principali modelli teorici presi in considerazione sono i seguenti:

- **Waterfall**: sequenza lineare con fasi ben separate;
- **Iterativo**: cicli successivi di revisione e miglioramento;
- **Incrementale**: rilascio progressivo di componenti funzionanti;
- **Agile**: pianificazione adattiva guidata da feedback continuo.

Nel dominio *code-first*$_G$ orientato alle API, è stato adottato un approccio incrementale e agile, coerente con il processo utilizzato dal team frontend, suddividendo il lavoro in piccoli incrementi in cui ogni ciclo aggiunge valore funzionale al software.

Durante lo sviluppo è stato mantenuto un confronto continuo tra i membri del team, con momenti periodici dedicati al monitoraggio dell'avanzamento, alla risoluzione dei problemi emersi e alla pianificazione delle attività successive.

Inoltre, è stato mantenuto un contatto costante con il tutor aziendale, aggiornando regolarmente lo stato di avanzamento e introducendo eventuali modifiche sulla base dei feedback ricevuti.

4.5.2 Ciclo teorico di un incremento

Un incremento di sviluppo può essere modellato come ciclo ripetibile:

1. analisi del requisito;
2. progettazione tecnica;

3. implementazione;
4. verifica (test e controlli qualitativi);
5. retrospettiva e pianificazione del ciclo successivo.

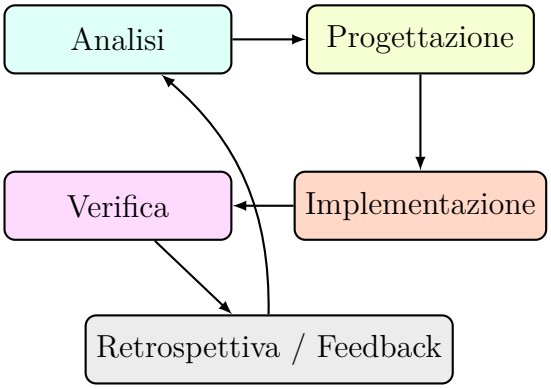

Figura 4.3: Ciclo teorico incrementale-iterativo di sviluppo

Come mostrato in Figura 4.3, l'incremento combina progettazione e verifica in un ciclo chiuso guidato dal feedback.

4.5.3 Strategia di verifica multilivello

La validazione è stata distribuita su più livelli, secondo una piramide concettuale di test: la sintesi visuale è riportata nella Figura 4.4.

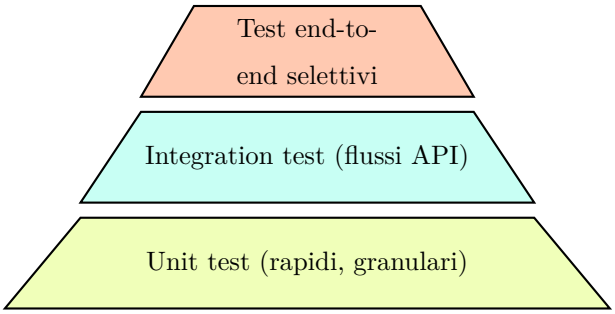

Figura 4.4: Piramide di test applicata al backend

In questa impostazione:

- i test unitari verificano componenti isolate e regole locali;
- i test di integrazione validano la cooperazione tra livelli applicativi;
- i test end-to-end confermano i flussi critici percepiti dall'utente.

4.5.4 Rischi tecnici e mitigazioni

In un processo incrementale orientato a servizi API, i rischi principali sono:

- **regressioni funzionali**: mitigate con test automatici su flussi critici;
- **degrado prestazionale**: mitigato tramite metriche di latenza e ottimizzazione query;
- **errori di autorizzazione**: mitigati con permessi granulari e test di accesso negato;
- **drift documentale**: mitigato con OpenAPI vicino al codice sorgente.

Un principio trasversale è la prevenzione anticipata: intercettare i segnali deboli (incremento degli errori, aumento della latenza, divergenza documentale) prima che si traducano in difetti sistemici.

In sintesi, il processo teorico proposto combina rigore tecnico e adattività: architettura modulare, sicurezza multilivello, verifica continua e decisioni guidate da evidenze osservative.

Capitolo 5

Progettazione e codifica

Questo capitolo descrive il lavoro svolto nella fase di progettazione e codifica del backend YesHomeGroup. L'obiettivo era realizzare un sistema di servizi *API*$_G$ robusto e manutenibile, capace di supportare le funzioni core del dominio (immobili, stanze, prezzi, prenotazioni, recensioni e utenti) garantendo sicurezza, osservabilità e tracciabilità.

5.1 Obiettivi, perimetro e vincoli

Il perimetro di lavoro ha incluso:

- definizione dell'architettura applicativa e della struttura dei moduli;
- progettazione e implementazione delle API *REST*$_G$ per le funzionalità di dominio;
- integrazione di sicurezza (*JWT*$_G$, *CSRF*$_G$, *RBAC*$_G$, *CORS*$_G$);
- gestione della persistenza con *ORM*$_G$ e query dedicate;
- osservabilità con logging e metriche Prometheus;
- supporto a notifiche email e upload immagini.

Il vincolo principale è stato mantenere un backend leggero e comprensibile, basato su Slim 4, con pipeline middleware esplicita e responsabilità ben separate.

5.2 Architettura generale e organizzazione del progetto

La codebase è organizzata per livelli funzionali, con responsabilità ben delimitate tra bootstrap, routing, middleware, controller, servizi e modelli dati. La separazione in cartelle facilita la navigazione e riduce l'accoppiamento tra componenti. L'impostazione generale è allineata ai principi dei micro-framework Slim e alle interfacce PSR per richieste/middleware *Slim Framework 4 Documentation*; *PSR-7: HTTP Message Interfaces*; *PSR-15: HTTP Server Request Handlers*.

Livello	**Path**	**Responsabilità**
Bootstrap	`src/Bootstrap/`	Avvio applicazione, configurazione *ORM*$_G$, logger, registry Prometheus.
Routing	`src/Routes/api.php`	Definizione delle rotte e composizione dei middleware.
Middleware	`src/Middleware/`	CORS, CSRF, JWT, permission check, logging, metriche.
Controllers	`src/Controllers/`	Gestione delle richieste *HTTP*$_G$, validazione input, orchestrazione servizi.
Services	`src/Services/`	Logica trasversale: email, immagini, JWT, logging applicativo.
Models	`src/Models/`	Entità *ORM*$_G$, relazioni e query di dominio.
Monitoring	`src/Monitoring/`	Definizione metrica e integrazione Prometheus.
Tests	`tests/`	Test unitari e di integrazione con DB in-memory.

Tabella 5.1: Struttura logica del backend YesHomeGroup

La Tabella 5.1 riassume la distribuzione delle responsabilità tra i principali livelli del backend.

5.2.1 Bootstrap e configurazione

L'entrypoint è definito in `public/index.php`, che esegue il bootstrap di Eloquent e carica le variabili d'ambiente tramite `.env`. Il file `src/Bootstrap/app.php` crea l'istanza Slim, registra l'endpoint `/swagger`, abilita l'error middleware e compone la pipeline con `MetricsMiddleware` e `LoggerMiddleware`. La configurazione DB è centralizzata in `config/database.php` e supporta driver differenti (MariaDB in produzione, SQLite in test). Il flusso risultante è rappresentato in Figura 5.1.

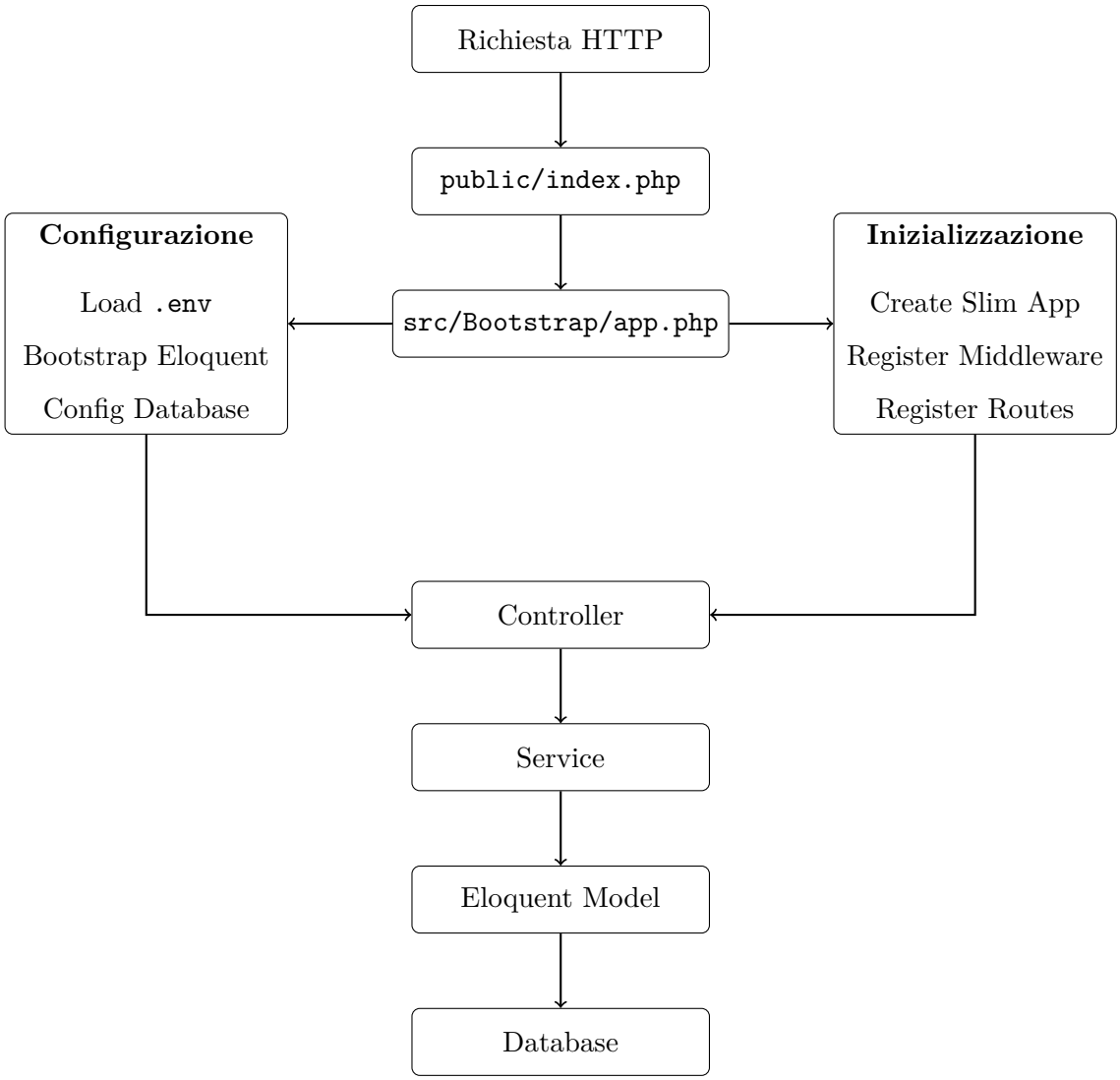

Figura 5.1: Flusso di bootstrap e inizializzazione dell'applicazione backend

5.2.2 Routing e composizione della pipeline

Le rotte sono dichiarate in `src/Routes/api.php`. La pipeline applicativa è composta in modo esplicito:

- CORS globale con `CorsMiddleware`, per gestire preflight *HTTP*$_G$ e header custom;
- protezione CSRF per le rotte mutanti tramite `CsrfMiddleware`;
- autenticazione *JWT*$_G$ per le aree protette;
- autorizzazione a grana fine con `PermissionMiddleware` e costanti in `Permission`.

Le rotte pubbliche includono `/login`, `/register`, `/metrics` e `/csrf`. Le rotte protette sono raggruppate e composte con middleware progressivi, garantendo un controllo deterministico dell'accesso. La sequenza completa di instradamento e controlli è mostrata nella Figura 5.2.

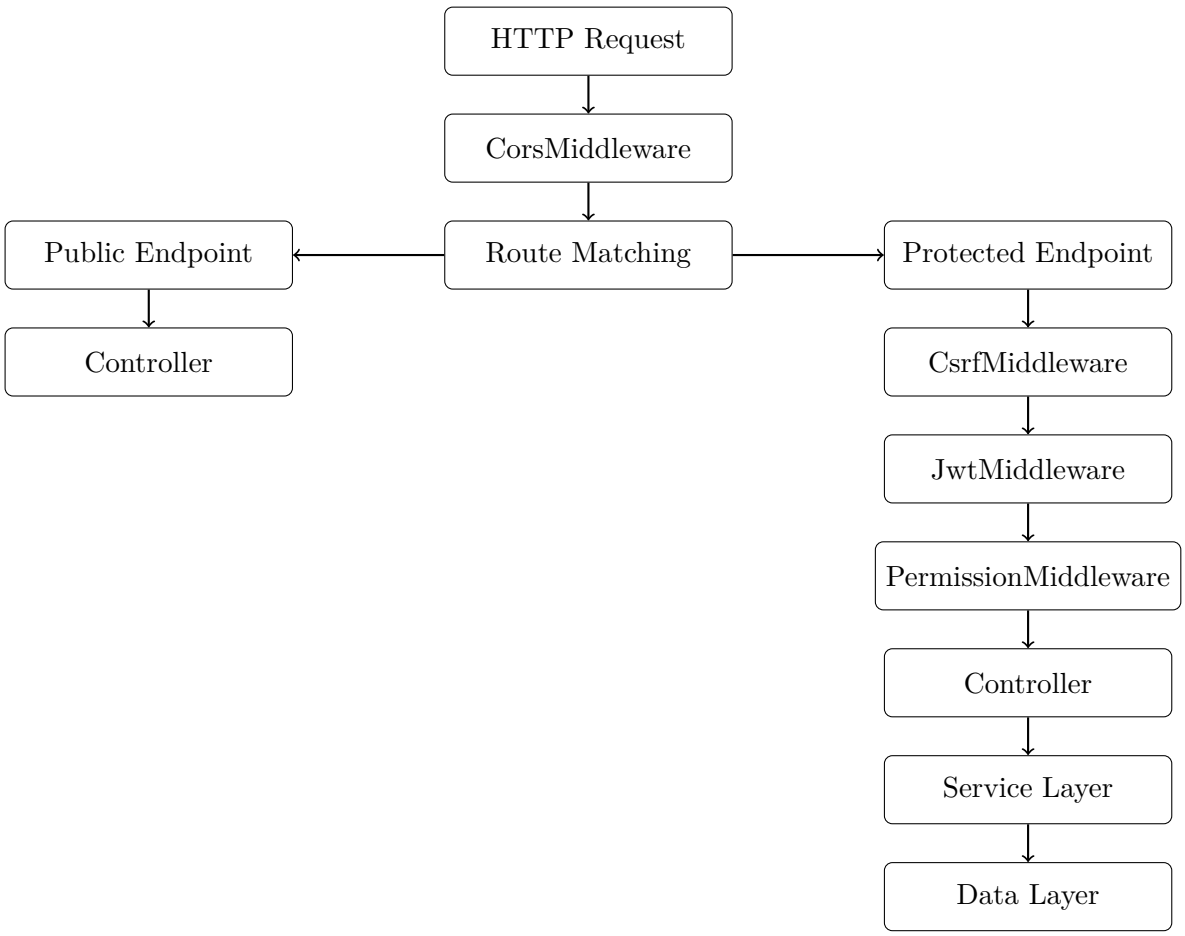

Figura 5.2: Composizione della pipeline di routing e dei middleware dell'applicazione

5.3 Progettazione delle API e documentazione

L'impostazione è di tipo resource-oriented, con endpoint coerenti per collezioni e singole risorse (ad esempio `/properties`, `/rooms`, `/bookings`). Le risposte utilizzano payload *JSON*$_G$ e codici di stato coerenti (400 per input non valido, 401 per mancanza di autenticazione, 403 per permessi insufficienti, 419 per CSRF non valido, 500 per errori inattesi). La scelta è coerente con la documentazione delle API HTTP di riferimento e con la specifica OpenAPI utilizzata per documentare i contratti MDN Web Docs contributors. *Cross-Origin Resource Sharing (CORS)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS (visitato il giorno 17/06/2026); *OpenAPI Specification*; *swagger-php*.

La documentazione *OpenAPI* $_G$ (OpenAPI Specification) viene mantenuta direttamente nel codice tramite attributi `OpenApi\Attributes`, con schema comune dichiarato in `src/OpenApiSpec.php`. L'endpoint `/swagger` genera dinamicamente la specifica, riducendo la divergenza tra implementazione e documentazione. La Figura 5.3 riassume il collegamento tra risorse REST, metodi controller e generazione della documentazione.

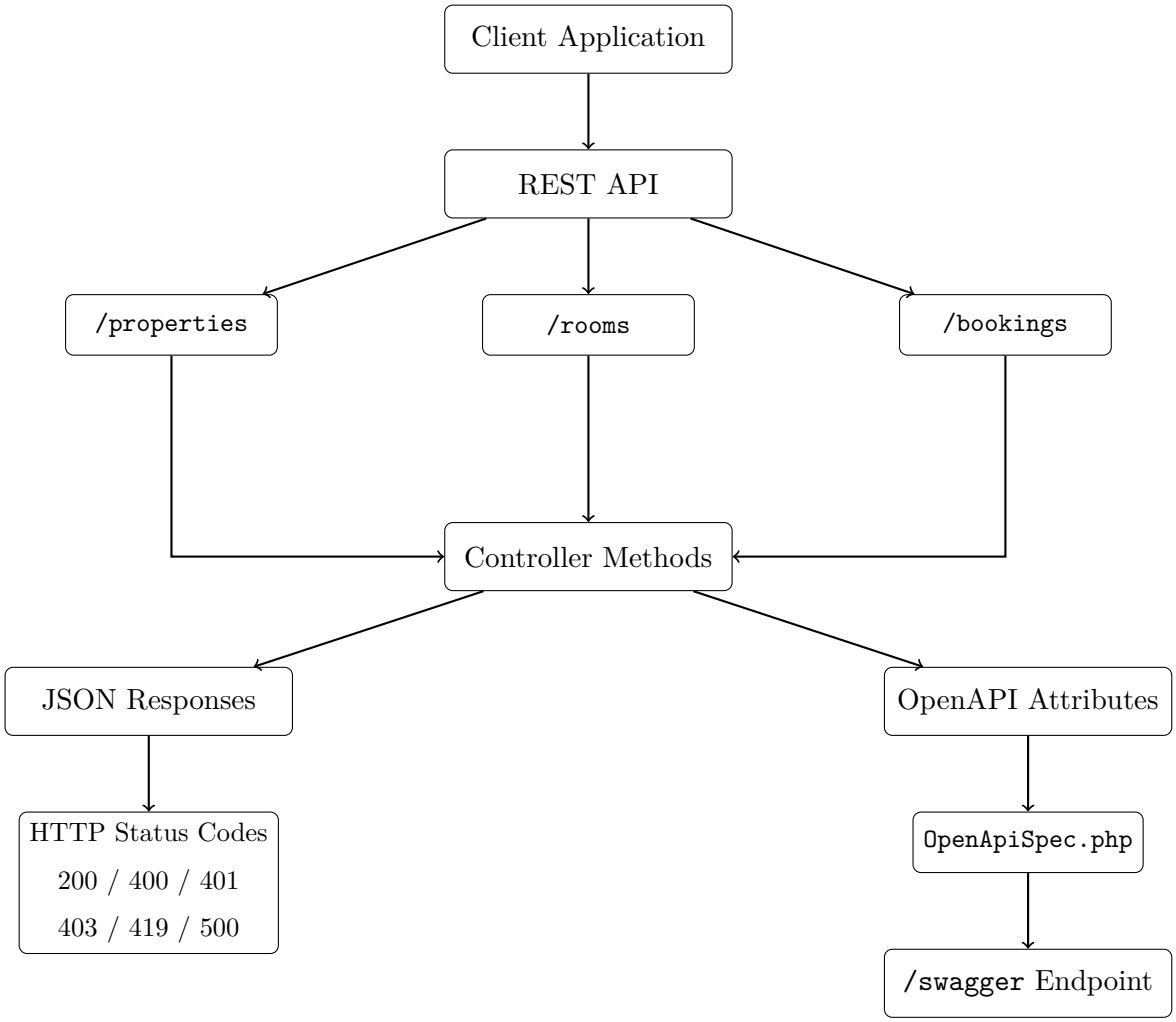

Figura 5.3: Architettura delle API REST e generazione automatica della documentazione OpenAPI.

5.4 Diagramma ER

Lo schema relazionale del database è stato progettato per rappresentare le principali entità del dominio applicativo e le loro relazioni, mantenendo un elevato grado di normalizzazione e garantendo l'integrità referenziale tramite chiavi

esterne. Per rendere più leggibile il modello, il diagramma è stato suddiviso in tre viste tematiche coerenti con le aree funzionali del sistema.

Il sistema di autenticazione e autorizzazione adotta un modello $RBAC_G$, in cui gli utenti sono associati ai ruoli attraverso la tabella ponte `user_role`, mentre i ruoli sono collegati ai permessi mediante `role_permission`. Nel dominio immobiliare, ogni proprietà può contenere più stanze, mentre ciascuna stanza appartiene a una sola proprietà.

Le prenotazioni collegano utenti e stanze mediante identificativi UUID e costituiscono il nucleo della logica di disponibilità. La gestione dei prezzi è separata in una tabella dedicata che permette di definire intervalli temporali differenti per ogni stanza. Le recensioni sono invece associate agli utenti e alle proprietà.

Le tabelle `featureables` e `images` sono relazioni polimorfiche: Mermaid permette di documentarle come entità intermedie, ma non rappresenta in modo perfetto la variabile `featureable_type`/`imageable_type`. Per questo motivo, nelle viste seguenti sono riportate come strutture condivise, lasciando alla descrizione testuale il chiarimento del vincolo polimorfico.

5.4.1 Utente/RBAC

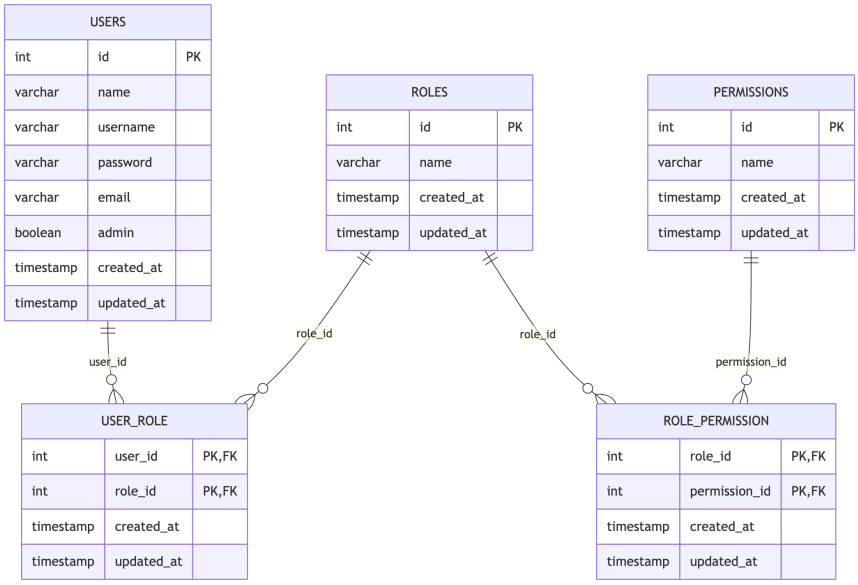

Figura 5.4: Vista ER del sottosistema di autenticazione e autorizzazione basato su RBAC.

La Figura 5.4 evidenzia la struttura di base del modello RBAC e le tabelle di giunzione utilizzate per ruoli e permessi.

Questa prima vista ER rappresenta il sottosistema di autenticazione e autorizzazione del sistema. L'entità `USERS` contiene le informazioni degli utenti registrati, con campi quali `username`, `email`, `password` (memorizzata in forma crittografata) e un flag `admin` per identificare gli amministratori. L'entità `ROLES` definisce i ruoli disponibili nel sistema (ad esempio guest, host, admin), mentre `PERMISSIONS` elenca i permessi granulari associabili ai ruoli (ad esempio `property:create`, `booking:edit`, `user:delete`).

Le tabelle di giunzione `USER_ROLE` e `ROLE_PERMISSION` implementano il modello RBAC (Role-Based Access Control). La relazione many-to-many tra utenti e ruoli consente di assegnare molteplici ruoli a un singolo utente, mentre la relazione tra ruoli e permessi consente di definire granularità fine nel controllo degli accessi. Questo approccio semplifica la gestione delle autorizzazioni e riduce la duplicazione di dati, poiché i permessi vengono definiti una sola volta e riutilizzati per più ruoli.

5.4.2 Property/Room

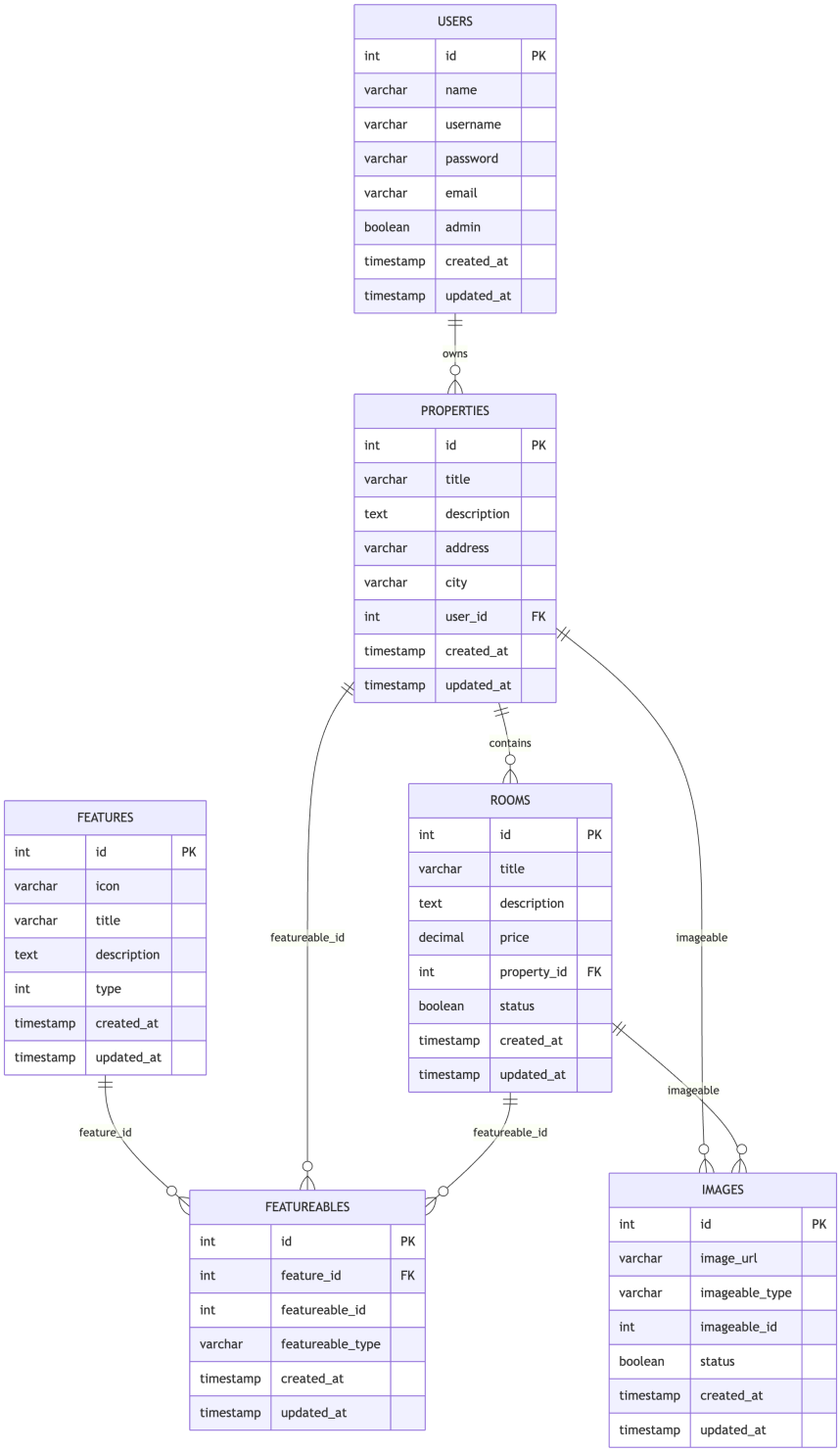

Figura 5.5: Vista ER delle entità immobiliari e delle relazioni polimorfiche riutilizzabili.

La Figura 5.5 mette in evidenza le relazioni tra proprietà, stanze e componenti polimorfici condivisi.

Questa seconda vista ER modella il dominio immobiliare del sistema. L'entità `PROPERTIES` rappresenta gli immobili pubblicati dagli host, con attributi descrittivi come titolo, descrizione, indirizzo e città. La relazione con `ROOMS` è di tipo uno-a-molti: una proprietà può contenere più stanze, mentre ogni stanza appartiene a una sola proprietà tramite chiave esterna `property_id`.

Le entità `FEATURES` e `IMAGES` sono rese riutilizzabili tramite relazioni polimorfiche. In particolare, `FEATUREABLES` collega una feature a diverse tipologie di entità (ad esempio stanza o proprietà) attraverso i campi `featureable_id` e `featureable_type`. Lo stesso principio è applicato alle immagini, consentendo una gestione uniforme dei media senza duplicare strutture. Questo approccio migliora estendibilità e manutenzione del modello dati.

5.4.3 Booking/Price/Log

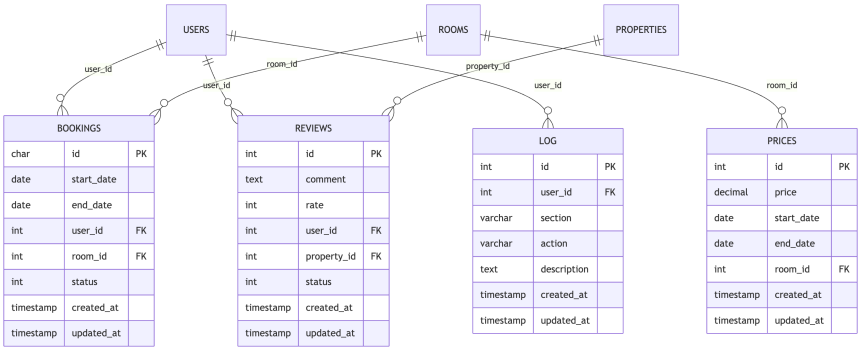

Figura 5.6: Vista ER delle prenotazioni, dei prezzi dinamici, delle recensioni e dei log applicativi.

La Figura 5.6 rappresenta le entità transazionali usate per disponibilità, pricing e audit operativo.

Questa terza vista ER raccoglie le entità transazionali e di tracciamento. `BOOKINGS` rappresenta il flusso di prenotazione, collegando utente e stanza con date di inizio/fine e stato operativo. Tale tabella è centrale per la verifica di disponibilità e per il controllo dei conflitti temporali tra prenotazioni sovrapposte.

`PRICES` introduce la tariffazione per intervalli temporali, permettendo prezzi diversi per la stessa stanza in periodi distinti. `REVIEWS` lega utenti e proprietà per gestire feedback e rating, mentre `LOG` conserva un audit trail delle operazioni applicative con riferimento all'utente, sezione, azione e descrizione. L'insieme di queste entità supporta sia la logica di business sia i requisiti di osservabilità e tracciabilità del sistema.

5.5 Relazioni ORM e logica applicativa

Per l'accesso ai dati è stato utilizzato Eloquent ORM, che consente di rappresentare le associazioni del modello relazionale attraverso relazioni orientate agli oggetti.

Le entità `User`, `Role` e `Permission` sono collegate mediante relazioni di Eloquent `belongsToMany`, implementando il modello RBAC direttamente a livello applicativo. Ogni `Property` espone una relazione `hasMany` verso `Room`, mentre ogni stanza utilizza una relazione `belongsTo` verso la proprietà di appartenenza.

Le prenotazioni sono associate sia agli utenti sia alle stanze mediante relazioni `belongsTo`; inoltre ogni stanza possiede molteplici intervalli di prezzo attraverso `hasMany`. Le immagini e le caratteristiche sono implementate mediante relazioni polimorfiche (`morphMany` e `morphToMany`), consentendo il riutilizzo della stessa struttura su entità differenti.

Oltre alla definizione delle relazioni, il livello ORM incorpora parte della logica di dominio. In particolare, il metodo `Booking::isAvailable` verifica l'assenza di sovrapposizioni temporali tra prenotazioni esistenti, mentre il calcolo dei prezzi utilizza query dedicate per determinare il listino valido in uno specifico intervallo temporale.

La Figura 5.7 mostra le principali relazioni tra i modelli Eloquent utilizzati nel progetto.

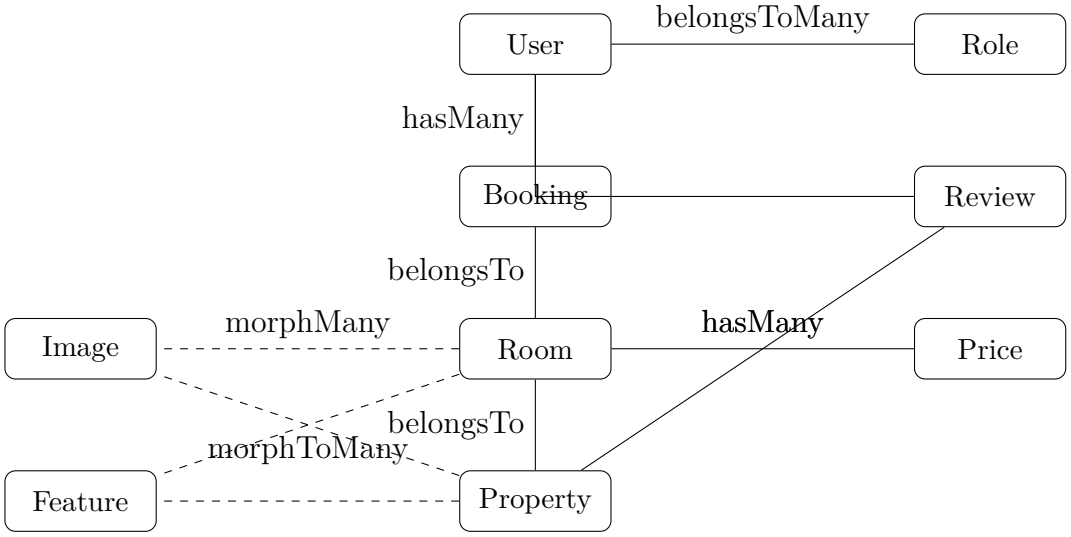

Figura 5.7: Principali relazioni definite nei modelli Eloquent ORM.

5.6 Servizi applicativi

Per evitare controller monolitici, alcune funzioni sono state estratte in servizi specializzati.

5.6.1 Gestione immagini

Il servizio `ImageService` gestisce l'upload, la compressione e la cancellazione delle immagini. Le immagini vengono salvate in cartelle distinte per entità (property/room) con filename univoci; in caso di errore, il file viene eliminato e non viene persistito nel database. La compressione utilizza la libreria GD con fallback sicuro se non disponibile. La pipeline operativa di questo servizio è illustrata in Figura 5.8.

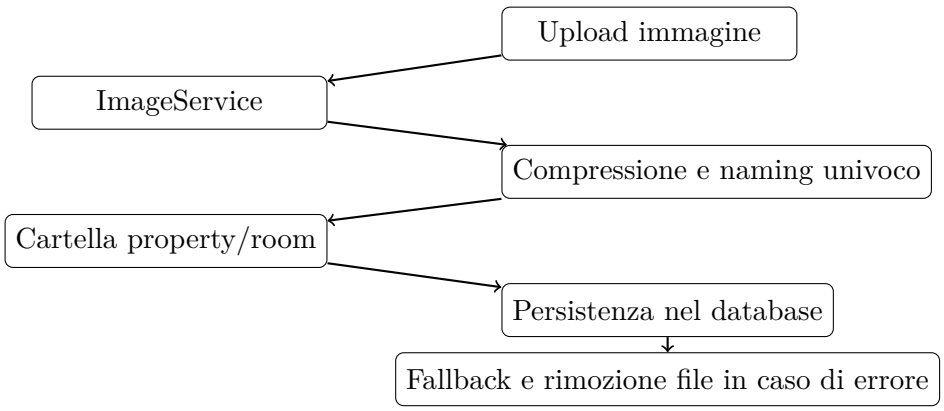

Figura 5.8: Flusso applicativo della gestione immagini tramite `ImageService`.

5.6.2 Notifiche email

Il servizio `MailerService` centralizza l'invio delle email (benvenuto, conferma prenotazione, aggiornamento prenotazione, notifiche recensione). La configurazione è delegata a variabili d'ambiente, con gestione esplicita del caso di mancata configurazione SMTP e metriche di successo/fallimento. Il flusso di invio e gestione esiti è sintetizzato in Figura 5.9.

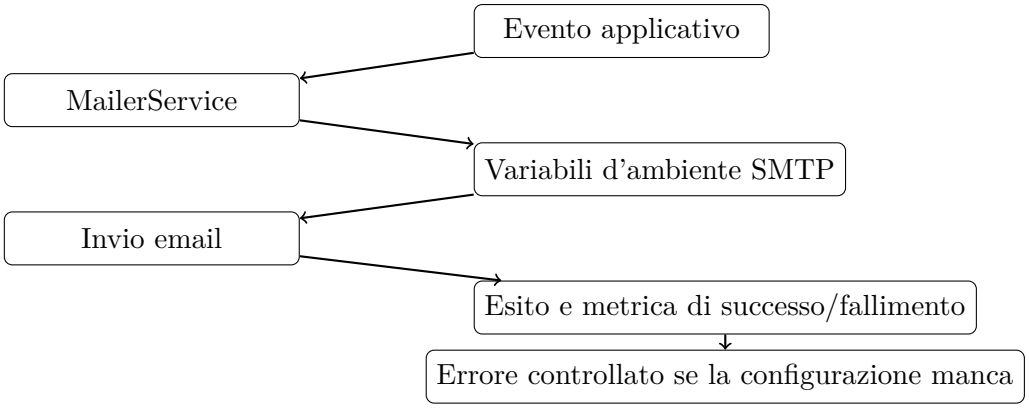

Figura 5.9: Flusso applicativo dell'invio email tramite `MailerService`.

5.6.3 Logging applicativo

Due livelli di logging sono utilizzati:

- logging tecnico via Monolog con rotazione file (`LoggerFactory`);
- logging applicativo via `LogService` su tabella `log` per audit di azioni utente.

La relazione tra logging tecnico e audit applicativo è mostrata in Figura 5.10.

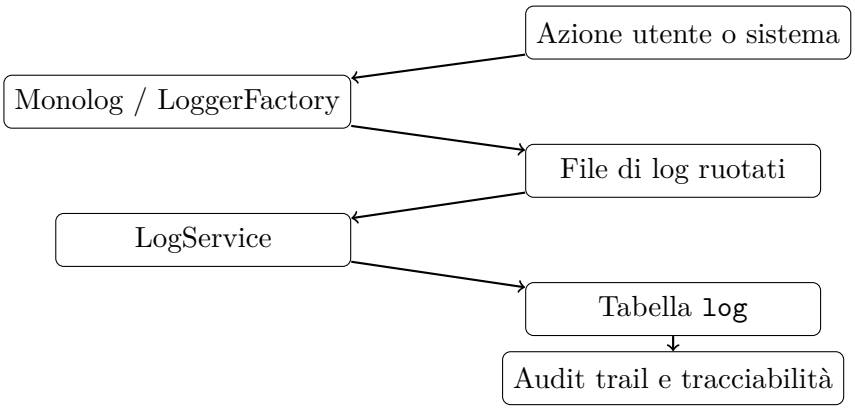

Figura 5.10: Flusso di logging tecnico e applicativo nel backend.

5.7 Sicurezza e controllo accessi

La sicurezza è implementata con una combinazione di meccanismi:

- *JWT*$_G$ per autenticazione stateless e propagazione dei claim utente;
- *CSRF*$_G$ per proteggere le operazioni mutanti; il token è distribuito tramite `/csrf` e cookie `XSRF-TOKEN`;
- *RBAC*$_G$ con permessi granulari per endpoint (es. `property:create`, `booking:admin_edit`);
- *CORS*$_G$ configurabile per integrare il client web e gestire il preflight.

Questa composizione evita dipendenze dirette tra controller e policy di sicurezza, migliorando la manutenibilità e la testabilità.

5.8 Osservabilità e metriche

L'osservabilità è fornita tramite:

- **Metriche Prometheus** per richieste *HTTP*$_G$, errori, prenotazioni, review, upload immagini e invio email;
- **Endpoint `/metrics`** per l'esposizione in formato Prometheus;

- **Storage flessibile** con fallback InMemory e opzione Redis per produzione.

Le metriche sono dichiarate centralmente in `src/Monitoring/Metrics.php`, garantendo consistenza semantica e naming uniforme.

5.9 Pattern e convenzioni di codifica

L'architettura segue un modello controller-service-model con pipeline middleware, allineato alle best practice Slim. Le convenzioni principali includono:

- autoload PSR-4 con namespace `App\`;
- stile PSR-12 e strumenti automatici di formatting e linting;
- codice orientato a responsabilità singola, con servizi dedicati alle funzionalità trasversali.

5.10 Problematiche riscontrate e soluzioni adottate

5.10.1 Sovrapposizioni temporali nelle prenotazioni

La gestione delle prenotazioni ha richiesto una validazione robusta delle date. Per evitare conflitti, `Booking::isAvailable` implementa una logica di overlap con fine esclusiva. In caso di conflitto, il sistema ritorna un errore 400 e incrementa la metrica `booking_conflict_total`.

5.10.2 Conflitti di prezzo e calendarizzazione

Il modello prezzi è basato su intervalli. Per prevenire sovrapposizioni indesiderate, `Price::findOverlaps` verifica i conflitti e, quando necessario, l'endpoint consente un override esplicito tramite flag `force`. Questo approccio mantiene coerenza dei dati senza bloccare casi d'uso amministrativi.

5.10.3 CSRF in contesto SPA

L'adozione di *CSRF*$_G$ in una *API*$_G$ è stata una criticità iniziale. La soluzione ha previsto:

- endpoint dedicato `/csrf` per generare token server-side;
- supporto a header `X-CSRF-TOKEN` e cookie `XSRF-TOKEN`;
- esclusione automatica per metodi safe (GET, HEAD, OPTIONS).

5.10.4 Parsing JSON e input non validi

Nel caso di body JSON malformati, alcuni controller producevano errori ambigui. La logica è stata rafforzata con controlli espliciti sull'input e messaggi d'errore chiari (es. in `UserController::updateProfile`).

5.10.5 Upload immagini e gestione spazio disco

Il caricamento di immagini ha richiesto:

- sanitizzazione dei nomi file;
- compressione automatica per ridurre l'occupazione su disco;
- cancellazione coerente tra filesystem e database.

In caso di errore, il servizio elimina i file parziali e restituisce uno stato per ogni immagine, evitando inconsistenze.

5.10.6 Configurazione email e failure controllati

La configurazione SMTP è esterna tramite variabili d'ambiente. Quando i parametri non sono presenti, il servizio ritorna un errore controllato e traccia il fallimento nelle metriche, evitando timeout applicativi.

5.10.7 Metriche in ambienti differenti

In assenza di Redis, il sistema utilizza uno storage InMemory per Prometheus. Il fallback è stato introdotto per garantire metriche anche in ambienti di sviluppo e test, mantenendo invariata la firma delle API.

5.11 Estratti di codice significativi

Qui ho raccolto alcuni estratti che, secondo me, rendono bene le parti più importanti del lavoro. Mostrano in modo diretto come il backend gestisce la disponibilità delle prenotazioni, l'autenticazione e la gestione dei prezzi. I frammenti riportati nei Listing 5.1, 5.2 e 5.3 evidenziano rispettivamente il controllo di overlap, la validazione del bearer token e la gestione dei conflitti di prezzo.

Codice 5.1: Verifica di disponibilità stanza con intervallo esclusivo

```
public static function isAvailable(int $roomId, string $startDate,
     string $endDate, ?string $excludeId = null): bool
{
    $query = self::where('room_id', $roomId);
    if ($excludeId !== null) {
        $query->where('id', '!=', $excludeId);
    }
    $query->where(function ($q) use ($startDate, $endDate) {
        $q->where('start_date', '<', $endDate)
          ->where('end_date', '>', $startDate);
    });
    return $query->exists();
}
```

Codice 5.2: Autenticazione stateless con JWT_G nel middleware

```
public function process(Request $request, Handler $handler):
    Response
{
    $authHeader = $request->getHeaderLine('Authorization');
```

```
4       if (!$authHeader || !preg_match('/Bearer\s(\S+)/', $authHeader
        , $matches)) {
5           return $this->unauthorized('Missing or invalid
        Authorization header');
6       }
7       $decoded = JWT::decode($matches[1], new Key($_ENV['JWT_SECRET'
        ], 'HS256'));
8       $request = $request->withAttribute('user', (array) $decoded);
9       return $handler->handle($request);
10  }
```

Codice 5.3: Creazione di un prezzo con controllo dei conflitti temporali

```
1   public function create(Request $request, Response $response):
        Response
2   {
3       $data = $request->getParsedBody();
4       $roomId = (int) ($data['room_id'] ?? 0);
5       $startDate = $data['start_date'] ?? '';
6       $endDate = $data['end_date'] ?? '';
7       $price = (float) ($data['price'] ?? 0);
8       $force = !empty($data['force']);
9
10      if (!$force) {
11          $conflicts = PriceModel::findOverlaps($roomId, $startDate,
         $endDate);
12          if (!empty($conflicts)) {
13              return $this->json($response, ['error' => 'Conflict,
        Price has already been set for the selected dates'], 400);
14          }
15      }
16
17      $id = PriceModel::create([
18          'room_id' => $roomId,
19          'start_date' => $startDate,
```

```
20          'end_date' => $endDate,
21          'price' => $price,
22      ]);
23
24      return $this->json($response, ['success' => 'Price added
        successfully', 'id' => $id], 201);
25  }
```

5.12 Sintesi

Il lavoro di progettazione e codifica ha prodotto un backend coerente con gli obiettivi del progetto: API ben documentate, modello dati espressivo, pipeline di sicurezza multilivello e osservabilità completa. La struttura modulare ha facilitato l'evoluzione del sistema e ha reso esplicite le responsabilità tecniche dei vari componenti.

Capitolo 6

Verifica e validazione

6.1 Strategia di verifica e validazione

La strategia di verifica è stata impostata su più livelli, con l'obiettivo di combinare feedback rapido e copertura funzionale. La piramide di test adottata è composta da:

- **test unitari**: validano componenti isolati (middleware, servizi, relazioni *ORM*$_G$);
- **test di integrazione**: verificano flussi completi sugli endpoint *HTTP*$_G$;
- **analisi statica**: controlli di stile e qualità del codice.

La validazione include inoltre verifiche manuali attraverso la documentazione *OpenAPI*$_G$ e il monitoraggio delle metriche Prometheus. Il framework metodologico è coerente con le pratiche descritte nella documentazione di PHPUnit, PHPCS, PHP-CS-Fixer e PHPStan *PHPUnit Manual*; PHPCSStandards, *PHP_CodeSniffer*; *PHP Coding Standards Fixer*; *PHPStan - PHP Static Analysis Tool*.

6.1.1 Ambiente di test e bootstrap

I test sono eseguiti su database SQLite in-memory, inizializzato da il file `tests/bootstrap.php`. Questo file crea lo schema, popola utenti e ruoli di base

e abilita le relazioni necessarie, garantendo isolamento e rapidità di esecuzione. Per i test di integrazione, `IntegrationTestCase` istanzia l'app Slim e prepara token *JWT*$_G$ e *CSRF*$_G$ per simulare richieste reali.

6.2 Test di unita

La suite di unit test copre componenti critici:

- **Middleware**: CORS e CSRF, con verifiche su preflight, header e gestione errori;
- **Services**: ImageService (compressione, gestione errori), MailerService (configurazione e failure controllati);
- **Models**: relazioni Eloquent (belongsTo, hasMany, morph).

Categoria	**Esempi**	**Obiettivo**
Middleware	`CorsMiddlewareTest`, `CsrfMiddlewareTest`	Verifica corretta gestione header e token.
Services	`ImageServiceTest`, `MailerServiceTest`	Validazione logica applicativa e failure controllati.
Models	`ModelRelationshipsTest`	Controllo consistenza delle relazioni *ORM*$_G$.

Tabella 6.1: Sintesi dei test unitari principali

La Tabella 6.1 riassume le aree prioritarie coperte dalla suite unitaria.

6.3 Test di integrazione

I test di integrazione coprono i controller principali (auth, booking, property, room, review, price, role). Le richieste sono costruite con helper dedicati che aggiungono token *JWT*$_G$ e *CSRF*$_G$, simulando l'uso reale da parte del frontend. I test verificano:

- corretta gestione dei casi validi e invalidi;
- codici di stato coerenti e messaggi di errore consistenti;
- differenze di permessi tra utenti standard e amministratori.

La costruzione delle richieste protette è sintetizzata nel Listing 6.1.

Codice 6.1: Helper per richieste protette nei test di integrazione

```
protected function createPostRequest(string $uri, array $body =
    [], ?string $token = null): ServerRequestInterface
{
    $token = $token ?? $this->adminToken;
    return (new ServerRequestFactory())
        ->createServerRequest('POST', $uri)
        ->withHeader('Authorization', 'Bearer ' . $token)
        ->withHeader('X-CSRF-TOKEN', $this->csrfToken)
        ->withHeader('Content-Type', 'application/json')
        ->withParsedBody($body);
}

```

6.4 Analisi statica e quality gate

Il progetto integra strumenti di controllo qualità eseguiti localmente e in pipeline *CI/CD*$_G$:

- **PHPUnit** per test unitari e di integrazione;
- **PHP_CodeSniffer** per conformità PSR-12;
- **PHP-CS-Fixer** per formattazione automatica;
- **PHPStan** per analisi statica.

Gli script composer definiscono comandi standardizzati (es. `composer test`, `composer phpcs`) che rendono il processo ripetibile e controllabile.

6.5 Validazione manuale e osservabilità

La validazione manuale è stata supportata da:

- endpoint `/swagger` per verificare la coerenza della documentazione *OpenAPI*$_G$;
- endpoint `/metrics` per monitorare traffico, errori e indicatori di dominio;
- controlli puntuali su flussi critici (registrazione, login, prenotazione, recensione).

Le metriche Prometheus hanno fornito un ulteriore livello di validazione runtime, evidenziando errori e anomalie in modo aggregato.

6.6 Problematiche emerse durante i test e soluzioni

6.6.1 Gestione CSRF nei test

L'uso di *CSRF*$_G$ richiede una sessione attiva e un token valido. Nei test, il problema è stato risolto generando il token in `IntegrationTestCase` e propagandolo via header `X-CSRF-TOKEN`.

6.6.2 Parsing del body nelle richieste DELETE

Alcuni controller leggono il body tramite `getParsedBody()`, altri tramite `getBody()`. Per evitare regressioni, i test di integrazione impostano sia il body parsed che lo stream raw, garantendo compatibilità con entrambe le varianti.

6.6.3 Differenze tra SQLite e MariaDB

L'uso di SQLite in-memory velocizza i test, ma alcune query specifiche (subquery per prezzi correnti) possono comportarsi in modo diverso rispetto a MariaDB. La soluzione è stata isolare tali logiche nei test e validarle con casi di integrazione mirati.

6.6.4 Dipendenze di ambiente (GD e SMTP)

Le feature che dipendono da librerie esterne (compressione immagini, invio email) possono fallire in ambienti non configurati. I test gestiscono questi casi con fallback controllati o skip condizionati.

6.7 Sintesi

La combinazione di test unitari, test di integrazione e quality gate ha garantito un livello di affidabilità adeguato all'obiettivo. Le verifiche manuali e le metriche Prometheus hanno completato il processo di validazione, offrendo feedback continuo durante lo sviluppo.

Capitolo 7

Conclusioni

7.1 Consuntivo finale

Il lavoro svolto ha portato alla realizzazione di un backend completo per YesHomeGroup, con API *REST*$_G$ documentate, sicurezza multilivello e osservabilità integrata. Le principali funzionalità consegnate includono:

- gestione di immobili, stanze, prezzi, prenotazioni e recensioni con *ORM*$_G$;
- autenticazione *JWT*$_G$ e controllo accessi *RBAC*$_G$;
- protezione *CSRF*$_G$ e configurazione *CORS*$_G$ per integrazione frontend;
- metriche Prometheus e logging applicativo;
- documentazione *OpenAPI*$_G$ generata dal codice.

7.2 Raggiungimento degli obiettivi

Gli obiettivi iniziali sono stati raggiunti con i seguenti risultati:

- API coerenti con il dominio e con contratti espliciti;
- pipeline middleware chiara e verificabile;
- modello dati normalizzato e relazioni consistenti;
- test unitari e di integrazione per i flussi critici;

- strumenti di qualità integrati nel workflow di sviluppo.

7.3 Conoscenze acquisite

Il progetto ha consolidato competenze su architetture backend moderne e strumenti di supporto:

- uso di Slim 4 per API *HTTP*$_G$ e pattern middleware;
- modellazione dati con *ORM*$_G$ Eloquent;
- autenticazione e autorizzazione con *JWT*$_G$ e *RBAC*$_G$;
- documentazione *OpenAPI*$_G$ e gestione di contratti;
- test automatici con PHPUnit e quality gate in *CI/CD*$_G$.

7.4 Possibili margini di miglioramento

Tra gli sviluppi futuri più rilevanti si evidenziano:

- rate limiting e caching per migliorare performance e resilienza;
- job asincroni per invio email e processing immagini;
- estensione delle metriche con alerting automatico;
- maggiore copertura di test end-to-end e scenari di concorrenza;
- migrazioni strutturate per evoluzioni schema DB.

7.5 Considerazioni personali

Durante il lavoro ho dovuto fare attenzione sia alla parte tecnica sia ai tempi di sviluppo. Le scelte sulla pipeline middleware e sul modello dati mi hanno aiutato a tenere il progetto ordinato, mentre logging e metriche sono state utili soprattutto quando qualcosa non funzionava come previsto.

7.6 Valutazione personale

Per me è stata un'esperienza molto utile, perchè mi ha fatto lavorare in modo più concreto su backend, test e gestione degli errori. Mi porto dietro un metodo più ordinato e una maggiore attenzione a sicurezza e tracciabilità.

Bibliografia

Testi

Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. a p. 24).

Sommerville, Ian. *Software Engineering*. 10ª ed. Pearson Education e Addison-Wesley, 2015 (cit. a p. 24).

Materiale didattico

Universita degli Studi di Padova. *Materiale didattico del corso*. Il materiale didattico presentato durante le lezioni e stato pubblicato progressivamente in formato digitale, collegato alle singole lezioni, congiuntamente alla registrazione della medesima lezione dell'anno accademico 2020/2021. 2021 (cit. a p. 24).

Sitografia

Composer Documentation. URL: https://getcomposer.org/doc/ (visitato il giorno 18/06/2026).

Eloquent: Getting Started. URL: https://laravel.com/docs/11.x/eloquent (visitato il giorno 17/06/2026) (cit. a p. 24).

firebase/php-jwt. URL: https://github.com/firebase/php-jwt (visitato il giorno 17/06/2026).

GitHub Actions Documentation. URL: https://docs.github.com/en/actions (visitato il giorno 18/06/2026) (cit. a p. 26).

MariaDB Documentation. URL: https://mariadb.com/docs/ (visitato il giorno 18/06/2026) (cit. a p. 24).

MDN Web Docs contributors. *Cross-Origin Resource Sharing (CORS)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS (visitato il giorno 17/06/2026) (cit. a p. 46).

Monolog - Logging for PHP. URL: https://github.com/Seldaek/monolog (visitato il giorno 18/06/2026) (cit. a p. 25).

OpenAPI Specification. URL: https://spec.openapis.org/oas/latest.html (visitato il giorno 17/06/2026) (cit. alle pp. 24, 46).

OWASP Foundation. *Cross Site Request Forgery (CSRF)*. URL: https://owasp.org/www-community/attacks/csrf (visitato il giorno 17/06/2026).

PHP Coding Standards Fixer. URL: https://github.com/PHP-CS-Fixer/PHP-CS-Fixer (visitato il giorno 18/06/2026) (cit. alle pp. 26, 61).

PHPCSStandards. *PHP_CodeSniffer*. URL: https://github.com/PHPCSStandards/PHP_CodeSniffer (visitato il giorno 18/06/2026) (cit. alle pp. 26, 61).

PHPStan - PHP Static Analysis Tool. URL: https://github.com/phpstan/phpstan (visitato il giorno 18/06/2026) (cit. alle pp. 26, 61).

PHPUnit Manual. URL: https://docs.phpunit.de/en/11.5/ (visitato il giorno 17/06/2026) (cit. alle pp. 26, 61).

Progetto Giovani Padova. *Coabitazione intergenerazionale nel Comune di Padova*. URL: https://www.progettogiovani.pd.it/animazione/coabitazione-intergenerazionale-nel-comune-di-padova/ (visitato il giorno 17/06/2026) (cit. a p. 1).

Prometheus Overview. URL: https://prometheus.io/docs/introduction/overview/ (visitato il giorno 17/06/2026) (cit. a p. 25).

PSR-12: Extended Coding Style Guide. URL: https://www.php-fig.org/psr/psr-12/ (visitato il giorno 17/06/2026).

PSR-15: HTTP Server Request Handlers. URL: https://www.php-fig.org/psr/psr-15/ (visitato il giorno 17/06/2026) (cit. alle pp. 23, 43).

PSR-4: Autoloader. URL: https://www.php-fig.org/psr/psr-4/ (visitato il giorno 17/06/2026).

PSR-7: HTTP Message Interfaces. URL: https://www.php-fig.org/psr/psr-7/ (visitato il giorno 17/06/2026) (cit. alle pp. 23, 43).

Slim Framework 4 Documentation. URL: https://www.slimframework.com/docs/v4/ (visitato il giorno 17/06/2026) (cit. alle pp. 23, 43).

swagger-php. URL: https://github.com/zircote/swagger-php (visitato il giorno 18/06/2026) (cit. alle pp. 24, 46).

Universita degli Studi di Padova. *Coabitazione intergenerazionale, al via la quarta edizione del progetto*. URL: https://www.unipd.it/news/coabitazione-intergenerazionale-al-la-quarta-edizione-del-progetto (visitato il giorno 18/06/2026) (cit. a p. 1).