Progetto di Secure Coding

Analisi delle vulnerabilità e hardening di
Hotel PMS

Property Management System per hotel

Diego Andruccioli

A.A. 2025/26

Sommario

Questa relazione documenta il processo di analisi della sicurezza e di *hardening* applicato a **Hotel PMS**, un sistema gestionale per la gestione alberghiera sviluppato come progetto personale e utilizzato come base per questo elaborato del corso di *Laboratorio di Sicurezza dei Sistemi e Privacy*.

Il sistema è un'applicazione enterprise a microservizi composta da 9 servizi backend (Java 21, Spring Boot 3.5, Spring Cloud 2025.0) e un frontend React 19, accessibili tramite un API Gateway centrale. La superficie di attacco è ampia e significativa: il sistema gestisce credenziali utente, sessioni JWT, dati personali di ospiti (PII), prenotazioni, fatture e report obbligatori per la Polizia di Stato.

Il documento segue l'approccio richiesto dalla traccia d'esame: partendo dalla baseline del codice originale (branch `pre-secure-coding`), identifica le vulnerabilità tramite analisi statica e threat modeling STRIDE, le mappa rispetto all'OWASP Top 10 2025, e descrive per ciascuna l'intervento di mitigazione applicato sul branch `main`, mostrando i diff rilevanti del codice prima e dopo l'hardening.

Ogni intervento è accompagnato dal riferimento al commit Git corrispondente e da un frammento della tabella di tracciamento del `THREAT_MODEL.md`, che evolve iterativamente con lo sviluppo.

Indice

1 Introduzione **5**
1.1 Descrizione del dominio applicativo . . . 5
1.2 Architettura del sistema . . . 5
1.3 Requisiti funzionali rilevanti per la sicurezza . . . 5
1.4 Metodologia di analisi . . . 6

2 Analisi della Baseline Insicura **7**
2.1 Panoramica delle vulnerabilità identificate . . . 7
2.2 Autenticazione e Gestione Password . . . 8
2.3 Validazione degli Input . . . 9
2.4 Autorizzazione e Controllo degli Accessi . . . 10
2.5 Gestione delle Sessioni . . . 11
2.6 Configurazioni di Sicurezza . . . 12

3 Implementazione del Secure Coding **14**
3.1 Autenticazione Sicura . . . 14
3.1.1 User Enumeration — Risposta Uniforme al Login [T-AUTH-01] . . . 14
3.1.2 Iniezione Header X-Auth via Client [T-GW-01] . . . 15
3.1.3 Token JWT in `localStorage` — Esposizione a XSS [T-FE-02] . . . 16
3.1.4 Brute Force sul Login — Assenza di Account Lockout [T-AUTH-02] 17
3.1.5 Sicurezza Hashing Password [T-AUTH-03] . . . 18
3.1.6 Refresh Token Rotation e Blacklist [T-AUTH-04] . . . 21
3.1.7 Cambio password non revoca le sessioni attive [T-AUTH-04 residuo] 25
3.1.8 Cambio password obbligatorio non applicato lato server [T-AUTH-06] 28
3.1.9 Riattivazione utente disattivato sempre fallita [T-AUTH-07] . . . 30
3.1.10 Refresh silenzioso escludeva /me, logout forzato ogni 15 minuti [T-AUTH-08] . . . 31
3.1.11 Cambio password volontario forzava un logout non necessario [T-AUTH-09] . . . 32
3.1.12 NullPointerException su login fallito senza clientIp — 401 diventava 500 [T-AUTH-14] . . . 34
3.1.13 Collasso della chiave di lockout Redis su clientIp nullo [T-AUTH-15] 35
3.2 Audit Log e Security Monitoring [Appendice A] . . . 37
3.3 Validazione, Sanificazione ed Encoding . . . 37
3.3.1 Input Validation su dati ospite [T-GST-02] . . . 37
3.3.2 Difesa contro Mass Assignment [T-GST-04] . . . 39
3.3.3 Server-side Price Validation [T-FB-02] . . . 40
3.3.4 Validazione importi pagamento [T-BILL-02] . . . 42

3.3.5 Validazione date di prenotazione (range cross-field) [T-RES-03] . . . 44
3.4 Autorizzazione e Controllo degli Accessi . . . 46
3.4.1 Strip Header X-Auth dal client [T-GW-01] . . . 46
3.4.2 IDOR su risorse ospite/hotel [T-GST-01, T-RES-02, T-BILL-01] . . 46
3.4.3 IDOR su ordini F&B — Accesso Cross-Hotel [T-FB-01] . . . 46
3.4.4 Double Booking Prevention [T-RES-01] . . . 49
3.5 Gestione Sicura delle Sessioni . . . 52
3.5.1 Cookie flags e JWT hardening [T-FE-02] / [T-AUTH-04] . . . 52
3.5.2 Protezione CSRF [T-GW-05] . . . 53
3.6 Configurazioni di Sicurezza (Difesa in Profondità) . . . 56
3.6.1 Security Headers HTTP [T-GW-03] . . . 56
3.6.2 Content Security Policy [T-FE-04] . . . 58
3.6.3 Output Encoding e Protezione XSS [T-FE-01] . . . 61
3.6.4 Route Guard basato sul Ruolo [T-FE-03] . . . 63
3.6.5 Protezione Actuator Endpoints [T-CFG-01] . . . 65
3.6.6 Rate Limiting — Estensione a Tutte le Route e Key Resolver Proxy-Aware [T-GW-02] . . . 67
3.6.7 CORS Misconfiguration — Whitelist Header Esplicita [T-GW-04] . . 69
3.7 Broken Access Control — Isolamento Multi-Tenant . . . 71
3.7.1 IDOR e Data Leak sul Guest Service [T-GST-01] [T-GST-03] . . . 71
3.7.2 IDOR sul Reservation Service [T-RES-02] . . . 73
3.7.3 IDOR sul Billing Service [T-BILL-01] . . . 75
3.7.4 Data Leak Cross-Tenant sul Report Finanziario Owner [T-BILL-04] 78
3.7.5 Denial of Service su Generazione PDF Fattura [T-BILL-05] . . . 79
3.7.6 Data Leak Cross-Tenant sul Pre-fill Check-in [T-STAY-06] . . . 80
3.7.7 IDOR e Tampering Cross-Tenant sul Catalogo Room Type [T-ROOM-02] . . . 81
3.7.8 Bypass di Autenticazione sul Canale SSE Realtime via Collisione HandlerMapping [T-GW-11] . . . 83
3.7.9 Verifica HMAC in Ingresso sul Gateway [T-GW-12] . . . 85
3.8 Gestione Segreti . . . 86
3.8.1 Rimozione segreti in chiaro [T-CFG-02] . . . 86
3.8.2 Regressione della guardia legale GDPR TULPS/fiscale [T-GST-06] . 89
3.9 Analisi SAST con CodeQL e Supply Chain Security [Appendice A] . . . 91
3.10 GAP-2 — Test E2E Attack-Path (Zero Coverage di Sicurezza) . . . 91
3.11 GAP-3 — fb-service InternalAuthFilter Struttura Inconsistente . . . 94
3.12 GAP-4 — Audit Log Non Aggregati (SIEM Assente) . . . 96
3.13 GAP-5 | T-GST-05 — GDPR Data Retention: Assenza di Policy sul Ciclo di Vita dei PII Ospite . . . 100
3.14 GAP-6 — Nessun Controllo Preventivo sull'Isolamento Multi-Tenant . . . 102

4 Gestione Autenticazione e Autorizzazione 105
4.1 Ridisegno del flusso di autenticazione . . . 105
4.2 Modello RBAC . . . 106
4.3 Gestione del ciclo di vita del JWT . . . 107
4.4 Protezioni contro attacchi comuni all'autenticazione . . . 108

A Appendice A — Implementazioni Avanzate (AI-assisted) 110
A.1 Autenticazione avanzata 110
A.2 Sicurezza inter-service avanzata 110
A.3 Configurazione e SDLC avanzati 120

B Appendice B — Utilizzo dell'IA e Revisione Critica 121
B.1 Metodologia 121
B.2 Episodio 1 — JWT storage: localStorage suggerito dall'AI 121
B.3 Episodio 2 — BCrypt: cost factor obsoleto per il 2025/26 122
B.4 Episodio 3 — IDOR: l'AI ha ignorato il contesto multi-tenant 122
B.5 Episodio 4 — CSRF: pattern incompatibile con SPA React 123
B.6 Episodio 5 — User enumeration: messaggi "utili per il debug" 124
B.7 Riepilogo della revisione critica 125

Capitolo 1

Introduzione

1.1 Descrizione del dominio applicativo

Hotel PMS (Property Management System) è un gestionale enterprise per strutture ricettive che supporta l'intero ciclo operativo di un hotel: dalla registrazione degli ospiti alla gestione delle prenotazioni, dal check-in/check-out alla fatturazione, fino al servizio di Food & Beverage del ristorante interno.

Il sistema è progettato in ottica multi-hotel: un singolo deployment gestisce più strutture, garantendo l'isolamento dei dati tramite il campo `hotel_id` propagato su tutti i servizi.

1.2 Architettura del sistema

Il backend è composto da 9 microservizi indipendenti, ciascuno con il proprio database PostgreSQL. Tutte le richieste client transitano attraverso l'**API Gateway** (:8080), che valida i token JWT e inietta gli header di autenticazione (`X-Auth-User`, `X-Auth-Role`, `X-Internal-Signature`) prima di instradare verso il servizio competente.

Servizio	Porta	Funzione
api-gateway	8080	Routing, JWT validation, rate limiting, CORS
config-service	8888	Configurazione centralizzata (segreti)
auth-service	8087	Autenticazione, JWT, gestione utenti
guest-service	8083	Anagrafica ospiti (PII)
inventory-service	8081	Gestione camere e disponibilità
reservation-service	8082	Prenotazioni
stay-service	8084	Check-in/check-out, report Alloggiati
billing-service	8085	Fatture e pagamenti
fb-service	8086	POS Food & Beverage

Tabella 1.1: Microservizi del sistema Hotel PMS

1.3 Requisiti funzionali rilevanti per la sicurezza

L'applicazione soddisfa tutti i requisiti minimi obbligatori indicati dalla traccia d'esame:

1. **Sistema di login e registrazione** — `auth-service`: form di autenticazione con JWT in cookie httpOnly, gestione ruoli (ADMIN, OWNER, RECEPTIONIST).
2. **Interazione persistente con database** — PostgreSQL per ogni microservizio, ORM JPA/Hibernate con query parametrizzate tramite Spring Data JPA.
3. **Gestione dei ruoli (RBAC)** — tre ruoli distinti con endpoint amministrativi riservati ad ADMIN, verificati lato server tramite Spring Security.
4. **Inserimento e visualizzazione dati utente** — campi testo libero su ospiti, note prenotazione, descrizioni ordini F&B.
5. **Risorse protette per proprietà** — ogni prenotazione/fattura/stay è legata a un ospite e un hotel; accesso verificato server-side tramite `hotel_id`.
6. **Gestione sessioni/stato** — JWT in cookie httpOnly con flag `Secure` e `SameSite=Strict`.

Funzionalità avanzate (bonus) presenti:

- Integrazione API esterna: report Alloggiati via Web Service Polizia di Stato (gestione sicura delle credenziali PS).
- Servizi esterni mockati: config-service come secret manager centralizzato.

1.4 Metodologia di analisi

L'analisi delle vulnerabilità segue la metodologia **STRIDE** di Microsoft, con mappatura rispetto all'**OWASP Top 10 2025**. Le vulnerabilità identificate sono documentate nel file `THREAT_MODEL.md` del repository e tracciate nella tabella di mitigazione che evolve con ogni commit di hardening.

Il branch `pre-secure-coding` rappresenta la **baseline** (codice originale), mentre il branch `main` contiene tutti gli interventi di messa in sicurezza.

Capitolo 2

Analisi della Baseline Insicura

Questo capitolo documenta le vulnerabilità rilevate nel codice del branch `presecure-coding` (baseline), organizzate per area tematica. Per ciascuna categoria viene presentata un'analisi del pattern insicuro con i relativi snippet. I dettagli completi di ogni intervento di mitigazione — con codice pre/post e riferimento al commit — sono nel Capitolo 3 (Implementazione del Secure Coding).

2.1 Panoramica delle vulnerabilità identificate

ID	OWASP	Descrizione	Impatto
T-AUTH-01	A07:2025	User Enumeration al login	ALTO
T-AUTH-02	A07:2025	Brute Force senza protezione	CRITICO
T-AUTH-03	A04:2025	Gestione password (hashing)	CRITICO
T-AUTH-04	A07:2025	Refresh Token senza revoca	ALTO
T-GW-01	A01:2025	Header X-Auth iniettabili	CRITICO
T-GW-03	A02:2025	Assenza security headers HTTP	MEDIO
T-GW-04	A02:2025	CORS misconfiguration: allowedHeaders wildcard	ALTO
T-GW-05	A01:2025	Assenza protezione CSRF	ALTO
T-GST-01	A01:2025	IDOR su ospiti (hotel_id)	CRITICO
T-GST-03	A01:2025	Multi-tenant leak PII ospiti	CRITICO
T-RES-01	A06:2025	Double booking (no date lock)	CRITICO
T-FB-02	A06:2025	Prezzi non ricalcolati server-side	CRITICO
T-CFG-01	A02:2025	Actuator esposti	CRITICO
T-CFG-02	A04:2025	Segreti in chiaro	CRITICO
T-FE-01	A05:2025	Assenza guard statico XSS (dangerouslySetInnerHTML)	ALTO
T-FE-03	A01:2025	Route guard solo lato client	MEDIO
T-FE-04	A02:2025	Assenza CSP	ALTO

Tabella 2.1: Riepilogo vulnerabilità identificate nella baseline

2.2 Autenticazione e Gestione Password

L'analisi della baseline ha rilevato tre vulnerabilità nell'area dell'autenticazione, classificabili OWASP A04:2025 e A07:2025, per un impatto complessivo da **alto** a **critico**.

T-AUTH-02 — Assenza di protezione al Brute Force Il metodo `login()` in `AuthServiceImpl` non manteneva alcun contatore di tentativi falliti né implementava alcun meccanismo di lockout applicativo. Un attaccante poteva eseguire un attacco a dizionario sull'endpoint `POST /api/v1/auth/login` senza limitazioni di sorta: il rate limiting del gateway (5 req/s per IP) costituiva l'unica difesa, facilmente aggirata distribuendo le richieste su più indirizzi IP (botnet, Tor).

AuthServiceImpl.java — assenza di account lockout (baseline)

```java
@Transactional(readOnly = true)
public AuthResponse login(LoginRequest request) {
    UserAccount user = userRepository.findByUsername(request.username())
        .orElseThrow(() -> new BadCredentialsException("INVALID_CREDENTIALS"));

    if (!passwordEncoder.matches(request.password(), user.getPasswordHash())) {
        throw new BadCredentialsException("INVALID_CREDENTIALS");
        // Nessun contatore, nessun lockout: tentativi illimitati
    }
    return buildAuthResponse(user);
}
```

T-AUTH-03 — Hashing password con cost factor insufficiente La baseline istanziava `BCryptPasswordEncoder` senza argomenti, usando il valore di default **cost 10** (circa 100 ms su hardware del 2013). Su GPU moderne questo valore permette di verificare centinaia di migliaia di hash al secondo, rendendo un attacco offline praticabile entro ore nel caso di furto del database.

Un secondo problema correlato: anche alzando il parametro in seguito, gli hash esistenti sarebbero rimasti a `cost 10` per tutti gli utenti che non avessero effettuato il logout e un nuovo login — senza un meccanismo di rehash progressivo, la protezione era retroattiva solo per i nuovi account.

SecurityConfig.java — cost factor 10 implicito (baseline)

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // strength=10 implicito, sotto soglia
        ↪ OWASP
}
```

T-AUTH-04 — Assenza di revoca del token (Refresh Token) La baseline emetteva un unico access token JWT con TTL configurabile ma senza alcun meccanismo di revoca anticipata. Non esisteva il concetto di *refresh token*: una volta emesso, il JWT rimaneva valido fino alla scadenza naturale anche dopo il logout esplicito, dopo una compromissione rilevata o in seguito a un cambio di credenziali. L'unica azione di "logout"

era cancellare il cookie lato client, ma il token continuava a essere accettato dal gateway per tutta la sua durata residua. Classificato OWASP A07 — Authentication Failures.

2.3 Validazione degli Input

La baseline presentava quattro aree di mancata validazione server-side, classificabili OWASP A05:2025 e A06:2025, che aprivano la via a injection, manipolazione dei prezzi e corruzione dei dati di prenotazione.

T-GST-02 — Assenza di vincoli di formato sui dati ospite I DTO di richiesta del `guest-service` (`GuestRequest`, `IdentityDocumentRequestDTO`) erano protetti solo con `@NotBlank` e `@Size`, ma privi di vincoli di formato sui campi testo libero. Un attaccante poteva inviare tag HTML o payload SQL nei campi `firstName`, `phone`, `address`, oppure una data di nascita nel futuro.

GuestRequest.java — nessun `@Pattern` (baseline)

```java
public record GuestRequest(
    @NotBlank @Size(max = 100) String firstName, // nessun @Pattern
    @NotBlank @Size(max = 100) String lastName, // nessun @Pattern
    @NotBlank @Email @Size(max = 150) String email,
    @Size(max = 20) String phone,            // nessun @Pattern
    @Size(max = 255) String address,         // nessun @Pattern
    @Size(max = 50) String city,             // nessun @Pattern
    @Size(max = 50) String country,          // nessun @Pattern
    LocalDate dateOfBirth) {                 // nessun @Past
}
// Payload valido inviabile: firstName = "<script>alert(1)</script>"
```

T-GST-04 — Mass Assignment nel mapper ospite `GuestMapper.toEntity()` non dichiarava esplicitamente i campi da ignorare durante il mapping da DTO a entità JPA. In assenza di `@Mapping(ignore=true)` su `id`, `hotelId`, `active`, `createdAt`, `updatedAt`, MapStruct avrebbe tentato di copiare qualsiasi campo omonimo presente nel DTO, aprendo la strada a una sovrascrittura dei metadati di sistema da parte del client (mass assignment).

T-RES-03 — Mancanza di validazione cross-field sulle date di prenotazione `ReservationRequest` non verificava che `checkOutDate` fosse successiva a `checkInDate`. Un client poteva inviare una prenotazione con date invertite o con `checkIn == checkOut` (zero-night stay), producendo una query di overlap detection semanticamente errata che poteva ignorare conflitti reali.

T-FB-02 — Prezzi degli ordini F&B controllati dal client `OrderItemRequest` accettava i campi `unitPrice` e `itemName` direttamente dalla richiesta HTTP. Il servizio si fidava del prezzo dichiarato dal client anziché consultare il catalogo server-side, consentendo a un utente di creare ordini con importi arbitrari (incluso zero o negativi) e di alterare il nome degli articoli. Classificato OWASP A06:2025 — Insecure Design.

T-BILL-02 — Assenza di validazione degli importi pagamento `PaymentRequest` non vincolava il campo `amount` con `@Digits` né `@Positive`, e `PaymentServiceImpl` non normalizzava l'importo prima di aggiungere il pagamento alla fattura. Importi negativi o con più di due decimali erano accettati, corrompendo il saldo e la logica di chiusura fattura.

2.4 Autorizzazione e Controllo degli Accessi

Questa categoria rappresenta l'area di rischio più estesa della baseline: sette vulnerabilità IDOR distribuite su cinque microservizi, più una rottura strutturale del meccanismo di isolamento multi-tenant a livello di gateway. Tutte classificate OWASP A01 — Broken Access Control.

T-GST-01 / T-GST-03 / T-RES-02 / T-BILL-01 / T-FB-01 — IDOR su cinque servizi I repository JPA di `guest-service`, `reservation-service`, `billing-service` e `fb-service` operavano senza scoping per `hotel_id`. Le query `findById()` e `findAll()` restituivano risorse di qualsiasi hotel, permettendo a un operatore dell'Hotel A, conoscendo o indovinando un UUID, di leggere e modificare ospiti, prenotazioni, fatture e ordini F&B dell'Hotel B.

Pattern IDOR — GuestServiceImpl.java (baseline)

```java
// INSICURO: lookup per solo UUID, nessun check ownership hotel
private Guest resolveGuest(UUID id) {
    return guestRepository.findById(id) // trova chiunque, qualsiasi hotel
        .orElseThrow(() -> new NotFoundException("GUEST_NOT_FOUND"));
}

// INSICURO: lista tutti gli ospiti di tutti gli hotel
public Page<GuestResponse> getAllGuests(Pageable pageable) {
    return guestRepository.findAll(pageable) // nessun filtro hotel_id
        .map(guestMapper::toResponse);
}
```

Lo stesso pattern era replicato in `ReservationServiceImpl`, `InvoiceServiceImpl` e `RestaurantOrderServiceImpl`: nessuno dei quattro servizi aveva `hotel_id` nella propria entità JPA né nelle query del repository.

T-GW-06 — Propagazione X-Auth-Hotel mancante end-to-end Anche con i fix IDOR applicati ai singoli servizi, l'isolamento multi-tenant avrebbe comunque fallito in produzione per una ragione strutturale più profonda: l'entità `UserAccount` in `auth-service` non possedeva il campo `hotel_id`, quindi `JwtService.generateToken()` non includeva il claim `hotelId` nel JWT emesso. Di conseguenza `AuthenticationFilter` nell'API Gateway non poteva iniettare l'header `X-Auth-Hotel` — perché il dato non era disponibile nel token — e i servizi interni ricevevano sempre l'header vuoto, restituendo 401 a ogni richiesta autenticata via gateway.

JwtService.java (auth-service) — claim hotelId assente (baseline)

```java
// INSICURO: il JWT non contiene il claim hotelId
public String generateToken(UserAccount user) {
    return Jwts.builder()
        .setSubject(user.getUsername())
        .claim("role", user.getRole().name())
        // hotelId mai aggiunto: l'isolamento multi-tenant non funziona
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + expiration))
        .signWith(key, SignatureAlgorithm.HS256)
        .compact();
}
```

T-FE-03 — Route guard solo lato client Il frontend React proteggeva le route amministrative (`/owner-dashboard`) nascondendo i link nella navigazione in base al ruolo memorizzato nello store Zustand. Tuttavia, `ProtectedRoute` non verificava il ruolo: un utente con ruolo `RECEPTIONIST` poteva accedere direttamente digitando l'URL nel browser, ricevendo dati o form riservati ad `ADMIN/OWNER`. Classificato OWASP A01 — Broken Access Control (frontend enforcement only).

T-RES-01 — Double Booking (assenza di lock concorrente) `ReservationServiceImpl.createReservation()` eseguiva un controllo di overlap sulle date di prenotazione, ma senza *pessimistic locking* sulla query. In condizioni di concorrenza (due richieste simultanee per la stessa camera e le stesse date) entrambe le transazioni potevano superare il controllo e inserire due prenotazioni in conflitto, violando il requisito di unicità operativa. Classificato OWASP A06:2025 — Insecure Design.

2.5 Gestione delle Sessioni

La baseline gestiva le sessioni tramite JWT in cookie `httpOnly` — scelta corretta per prevenire l'accesso JavaScript al token — ma presentava due lacune rilevanti nella gestione del ciclo di vita del token e nella protezione dalle richieste cross-site.

T-AUTH-04 — Refresh token senza rotazione né blacklist Redis Il sistema non implementava il meccanismo di *refresh token*: il solo access token JWT, una volta emesso, era accettato dal gateway per tutta la sua durata (configurabile, default 15 min) anche dopo il logout. Non esisteva alcun identificatore univoco (`jti`) per tracciare il token, né una blacklist su Redis per invalidarlo anticipatamente. Di conseguenza:

- Un logout non revocava effettivamente la sessione: il token rubato prima del logout restava valido.
- Un cambio di password o ruolo non invalidava i token emessi in precedenza.
- Uno scenario di *token theft* non aveva rimedio applicativo immediato.

T-GW-05 — Assenza di protezione CSRF L'API Gateway non implementava alcun meccanismo esplicito di protezione CSRF. Il flag `SameSite=Strict` sul cookie JWT offre una prima difesa, ma non è sufficiente: browser datati, proxy aziendali o subdomain takeover possono violarne il rispetto. In assenza di un token CSRF verificato server-side, un sito malevolo poteva indurre il browser dell'utente autenticato a inviare richieste mutanti (`POST`, `PUT`, `DELETE`) verso il sistema, sfruttando l'invio automatico del cookie da parte del browser.

api-gateway (baseline) — nessun filtro CSRF

```
// INSICURO: il gateway accetta qualsiasi richiesta con cookie JWT valido,
// indipendentemente dall'origine della richiesta.
// Non esiste alcun GlobalFilter che validi un CSRF token.
// Un sito malevolo puo' scatenare:
//   POST https://hotel.example.com/api/v1/reservations
//   Cookie: jwt=<token valido> (inviato automaticamente dal browser)
// La richiesta viene eseguita senza che il server verifichi l'origine.
```

2.6 Configurazioni di Sicurezza

La baseline presentava sei categorie di misconfiguration, alcune con impatto critico (segreti in chiaro, actuator esposti), altre con impatto medio-alto (rate limiting parziale, assenza di header HTTP, CORS permissivo, mancanza di CSP). Classificate prevalentemente OWASP A02:2025 — Security Misconfiguration.

T-CFG-01 — Endpoint `/actuator` esposti sulla porta applicativa Tutti e 9 i microservizi pubblicavano gli endpoint Spring Boot Actuator (`/actuator/health`, `/actuator/prometheus`, `/actuator/info`) sulla stessa porta dell'API (`8080`–`8087`, `8888`), a sua volta esposta nel `docker-compose.yml`. `/actuator/prometheus` rivelava metriche interne del processo JVM (thread pool, connection pool, circuit breaker state) preziose per un attaccante in fase di *reconnaissance*. `/actuator/health` con `show-details: always` esponeva lo stato dei sottosistemi e i dettagli di connessione a database e Redis.

T-CFG-02 — Segreti in chiaro nei file versionati Due categorie di segreti erano presenti in chiaro nel repository:

1. Il fallback hardcoded nella proprietà `jwt.secret` in `auth-service.yml` e in `@Value` di `api-gateway/JwtUtil.java`: entrambi contenevano la stringa Base64 nota `bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==`. Se la variabile d'ambiente `JWT_SECRET` non veniva impostata, il servizio avviava silenziosamente con questo segreto debole, permettendo a chiunque conosca il repository di forgiare JWT validi.

2. `POSTGRES_PASSWORD` hardcoded a `password` nel `docker-compose.yml`: password identica per tutti gli ambienti, senza possibilità di rotazione senza modifica al codice sorgente.

auth-service.yml + JwtUtil.java — fallback hardcoded (baseline)

```
# auth-service.yml: fallback noto, committato nel repository
jwt:
  secret: "${JWT_SECRET:
      ↪ bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==}"
```

```
// JwtUtil.java (api-gateway): stesso fallback anche nel gateway
@Value("${jwt.secret:
    ↪ bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==}")
private String secret;
```

T-CFG-03 — Config Server senza autenticazione Il `config-service` (Spring Cloud Config) distribuiva la configurazione centralizzata — inclusi segreti, credenziali e chiavi — senza alcuna protezione HTTP Basic. Qualsiasi processo raggiungibile sulla porta `8888` della rete Docker poteva ottenere la configurazione completa di ogni microservizio con una semplice richiesta GET.

T-GW-02 — Rate limiting applicato solo alla route `/auth` Nella baseline il `RequestRateLimiter` era configurato esclusivamente sulla route di autenticazione (`/api/v1/auth/**`). Le sei route di business (`/guests`, `/reservations`, `/stays`, `/invoices`, `/fb`, `/rooms`) non avevano alcun limite di frequenza, esponendo i servizi interni ad attacchi di enumerazione e a scenari di denial-of-service applicativo. Inoltre, il `KeyResolver` leggeva l'indirizzo TCP remoto (`RemoteAddress`), risultando inutile per deployment dietro proxy/load balancer che presentano un unico IP al gateway.

T-GW-03 / T-GW-04 — Assenza di security headers HTTP e CORS permissivo Nessuno dei microservizi né il gateway aggiungeva gli header di sicurezza HTTP fondamentali: `Strict-Transport-Security`, `X-Content-Type-Options`, `X-Frame-Options`, `Referrer-Policy` e `Permissions-Policy` erano assenti da tutte le risposte. La configurazione CORS usava `allowedHeaders: "*"` (wildcard), trasmettendo qualsiasi header client al backend senza whitelist esplicita.

T-FE-04 — Assenza di Content Security Policy La SPA React servita da Nginx non definiva alcuna `Content-Security-Policy`. In assenza di CSP, un XSS riuscito avrebbe potuto caricare script arbitrari da domini esterni, estrarre dati e contattare server C&C senza restrizioni. Ugualmente assente era qualsiasi meccanismo di prevenzione statica del `dangerouslySetInnerHTML` lato frontend.

T-AUTH-05 / T-STAY-02 / T-BILL-03 — Assenza di audit logging Nella baseline nessun servizio emetteva log strutturati per gli eventi sensibili: tentativi di login (riusciti e falliti), check-in/check-out e transazioni di pagamento avvenivano senza traccia applicativa. In uno scenario di incidente o frode, l'assenza di audit trail avrebbe reso impossibile ricostruire la sequenza degli eventi. Classificato OWASP A09:2025 — Security Logging & Alerting Failures.

Capitolo 3

Implementazione del Secure Coding

3.1 Autenticazione Sicura

title=Mitigazioni presenti nella baseline (branch `pre-secure-coding`)

Le vulnerabilità **T-AUTH-01**, **T-GW-01** e **T-FE-02** risultano già mitigate nel codice del branch `pre-secure-coding` (baseline). Tali protezioni sono state introdotte consapevolmente durante la fase di sviluppo iniziale, non come hardening successivo. Le sezioni seguenti documentano il pattern insicuro che è stato evitato, a dimostrazione della consapevolezza del rischio durante la progettazione.

3.1.1 User Enumeration — Risposta Uniforme al Login [T-AUTH-01]

Descrizione Un endpoint di login che restituisce messaggi di errore differenziati (es. `"USER_NOT_FOUND"` vs `"INVALID_PASSWORD"`) consente a un attaccante di enumerare gli username validi tramite una scansione automatizzata. Classificato OWASP A07 — Authentication Failures.

Pattern Vulnerabile Evitato — AuthServiceImpl.java

```java
// INSICURO: messaggi diversi rivelano l'esistenza dell'account
public AuthResponse login(LoginRequest request) {
    UserAccount user = userRepository.findByUsername(request.username())
        .orElseThrow(
            () -> new BadCredentialsException("USER_NOT_FOUND")); // leak!

    if (!passwordEncoder.matches(request.password(), user.getPasswordHash())) {
        throw new BadCredentialsException("INVALID_PASSWORD"); // diverso -> enum
    }
    ...
}
```

Mitigazione applicata (baseline) `AuthServiceImpl.login()` utilizza lo stesso messaggio `"INVALID_CREDENTIALS"` sia quando l'utente non esiste sia quando la password è errata, rendendo impossibile distinguere i due casi dall'esterno.

Codice Sicuro — AuthServiceImpl.java (branch main)

```java
// SICURO: stesso messaggio in entrambi i casi
public AuthResponse login(LoginRequest request) {
    UserAccount user = userRepository.findByUsername(request.username())
        .orElseThrow(
            () -> new BadCredentialsException("INVALID_CREDENTIALS"));

    if (!passwordEncoder.matches(request.password(), user.getPasswordHash())) {
        throw new BadCredentialsException("INVALID_CREDENTIALS"); // identico
    }
    ...
}
```

Riferimento Presente nella baseline — `auth-service/AuthServiceImpl.java`.

3.1.2 Iniezione Header X-Auth via Client [T-GW-01]

Descrizione In un'architettura a microservizi, i servizi interni ricevono dal gateway gli header `X-Auth-User` e `X-Auth-Role` per identificare il chiamante senza ri-validare il JWT. Un servizio che accetti tali header senza verificarne l'origine è vulnerabile a impersonation: un client che raggiunga direttamente la porta interna del servizio può iniettare `X-Auth-Role: ADMIN` e ottenere privilegi massimi. Classificato OWASP A01 — Broken Access Control.

Pattern Vulnerabile Evitato — filtro servizi interni

```java
// INSICURO: il servizio trust gli header senza verifica di origine
String username = request.getHeader("X-Auth-User");
String role   = request.getHeader("X-Auth-Role");
// Chiunque raggiunga direttamente il servizio puo' iniettare ROLE_ADMIN
SecurityContextHolder.getContext().setAuthentication(
    new UsernamePasswordAuthenticationToken(
        username, "", List.of(new SimpleGrantedAuthority("ROLE_" + role))));
```

Mitigazione applicata (baseline) Il gateway firma ogni richiesta con `HMAC-SHA256(secret, "username:role")` e inietta la firma in `X-Internal-Signature`. L'`InternalAuthFilter` presente in ogni microservizio ricalcola la firma e la confronta con `MessageDigest.isEqual` (confronto constant-time, immune a timing attack) prima di popolare il `SecurityContext`. Senza il segreto HMAC condiviso è impossibile forgiare una firma valida.

Codice Sicuro — InternalAuthFilter.java (tutti i servizi)

```java
// SICURO: verifica HMAC-SHA256 prima di fidarsi degli header
private boolean isSignatureValid(String username, String role, String sig) {
    String expected = computeHmac(username, role);
    // Confronto constant-time: previene timing attack
    return MessageDigest.isEqual(
        expected.getBytes(StandardCharsets.UTF_8),
```

```
        sig.getBytes(StandardCharsets.UTF_8));
}

// In doFilterInternal():
if (!isSignatureValid(username, role, signature)) {
    rejectRequest(response, "Invalid internal request signature"); // 401
    return;
}
```

Riferimento Presente nella baseline — `guest-service/.../InternalAuthFilter.java` (replicato in tutti i microservizi).

3.1.3 Token JWT in `localStorage` — Esposizione a XSS [T-FE-02]

Descrizione Memorizzare il token di sessione in `localStorage` lo rende leggibile da qualsiasi script JavaScript eseguito nella pagina, incluso codice iniettato via XSS. Un attaccante può estrarlo con `localStorage.getItem('jwt')` ed eseguire richieste autenticate su server arbitrari. Classificato OWASP A04:2025 — Cryptographic Failures / A07.

Pattern Vulnerabile Evitato — gestione token frontend

```
// INSICURO: token in localStorage, leggibile da XSS
const response = await axios.post('/api/v1/auth/login', credentials);
localStorage.setItem('jwt', response.data.token); // esposto a JS
axios.defaults.headers.common['Authorization'] =
    `Bearer ${localStorage.getItem('jwt')}`;
```

Mitigazione applicata (baseline) Il server imposta il JWT in un cookie `httpOnly; Secure; SameSite=Strict`. Il flag `httpOnly` lo rende inaccessibile a JavaScript; `Secure` ne impedisce la trasmissione in chiaro; `SameSite=Strict` blocca l'invio cross-site. Il frontend React non gestisce mai il valore del token.

Codice Sicuro — AuthController.java (branch `main`)

```
// SICURO: cookie httpOnly, invisibile a JavaScript
private ResponseCookie createCookie(String value, int maxAge) {
    return ResponseCookie.from("jwt", value)
        .httpOnly(true)     // inaccessibile a JavaScript
        .secure(true)       // solo HTTPS
        .sameSite("Strict") // blocco invio cross-site
        .path("/")
        .maxAge(maxAge)
        .build();
}
// Il frontend non tocca mai il token: il browser lo gestisce in automatico
```

Riferimento Presente nella baseline — `auth-service/AuthController.java`.

3.1.4 Brute Force sul Login — Assenza di Account Lockout [T-AUTH-02]

Descrizione L'assenza di un meccanismo di account lockout consente attacchi a dizionario o brute force sull'endpoint di login senza limitazioni applicative. Il rate limiting del gateway (5 req/s per IP) è insufficiente contro attacchi distribuiti da IP multipli. Classificato OWASP A07 — Authentication Failures.

La baseline non implementava alcun contatore di tentativi falliti:

Codice Vulnerabile — AuthServiceImpl.java (baseline, branch pre-secure-coding)

```java
@Transactional(readOnly = true)
public AuthResponse login(LoginRequest request) {
    UserAccount user = userRepository.findByUsername(request.username())
        .orElseThrow(() -> new BadCredentialsException("INVALID_CREDENTIALS"));

    if (!passwordEncoder.matches(request.password(), user.getPasswordHash())) {
        throw new BadCredentialsException("INVALID_CREDENTIALS");
        // Nessun contatore, nessun lockout:
        // un attaccante puo' provare password illimitatamente
    }

    return new AuthResponse(
        jwtService.generateToken(user.getUsername(), user.getRole()));
}
```

Mitigazione applicata Il metodo `login()` è stato esteso con una politica di lockout progressivo:

- Dopo **5 tentativi falliti consecutivi** il campo `locked_until` viene impostato a `now + 15 minuti`.
- Se l'account è bloccato, viene lanciata `AccountLockedException` (HTTP 429 Too Many Requests) *prima* del controllo password, evitando timing leak sulla validità della password.
- Un login riuscito resetta il contatore e cancella il lock.
- I valori (soglia e durata) sono costanti named, senza magic numbers.

La Flyway migration `V2__add_login_lockout.sql` aggiunge le colonne `failed_attempts` e `locked_until` alla tabella `user_account`.

Codice Sicuro — AuthServiceImpl.java (main)

```java
private static final int MAX_FAILED_ATTEMPTS = 5;
private static final Duration LOCKOUT_DURATION = Duration.ofMinutes(15);

@Transactional
public AuthResponse login(LoginRequest request) {
    UserAccount user = userRepository.findByUsername(request.username())
        .orElseThrow(() -> new BadCredentialsException("INVALID_CREDENTIALS"));
```

```java
    // 1. Verifica lockout PRIMA del controllo password (no timing leak)
    if (user.getLockedUntil() != null
            && Instant.now().isBefore(user.getLockedUntil())) {
        throw new AccountLockedException("ACCOUNT_TEMPORARILY_LOCKED"); // 429
    }

    // 2. Verifica password
    if (!passwordEncoder.matches(request.password(), user.getPasswordHash())) {
        int newAttempts = user.getFailedAttempts() + 1;
        user.setFailedAttempts(newAttempts);
        if (newAttempts >= MAX_FAILED_ATTEMPTS) {
            user.setLockedUntil(Instant.now().plus(LOCKOUT_DURATION));
        }
        userRepository.save(user);
        throw new BadCredentialsException("INVALID_CREDENTIALS");
    }

    // 3. Login riuscito: reset contatore e lock
    if (user.getFailedAttempts() > 0) {
        user.setFailedAttempts(0);
        user.setLockedUntil(null);
        userRepository.save(user);
    }
    return new AuthResponse(
        jwtService.generateToken(user.getUsername(), user.getRole()));
}
```

V2___add_login_lockout.sql — Flyway migration

```sql
ALTER TABLE user_account
    ADD COLUMN IF NOT EXISTS failed_attempts INTEGER NOT NULL DEFAULT 0,
    ADD COLUMN IF NOT EXISTS locked_until TIMESTAMPTZ;
```

Test Aggiunti 4 test in `AuthServiceImplTest`: `loginThrowsWhenAccountIsLocked`, `loginIncrementsFailedAttemptsOnWrongPassword`, `loginLocksAccountAfterMaxFailedAttempts`, `loginResetsCounterOnSuccess`.

Riferimento Commit: `security(auth): add account lockout for brute-force protection (T-AUTH-02)`

3.1.5 Sicurezza Hashing Password [T-AUTH-03]

Descrizione L'algoritmo BCrypt adotta un *cost factor* (o *strength*) che determina il numero di iterazioni dell'algoritmo: un valore troppo basso rende il calcolo dell'hash banalmente veloce su hardware moderno, abbattendo il tempo necessario per un attacco brute-force su hash rubati. La configurazione originale usava `new BCryptPasswordEncoder()` senza argomenti, che corrisponde al valore di default **10** (circa 100 ms su hardware del 2013). L'OWASP Authentication Cheat Sheet raccomanda un cost factor minimo di **10** e preferibilmente **12** per hardware contemporaneo (≈400 ms per hash). Classificato OWASP A04:2025 — Cryptographic Failures.

Un secondo problema correlato è l'**assenza di rehash progressivo**: anche dopo aver alzato il cost factor, gli hash esistenti nel database rimangono con il vecchio valore finché non vengono aggiornati esplicitamente. Senza un meccanismo di aggiornamento automatico, la protezione è retroattiva solo per i nuovi utenti.

Codice Vulnerabile — SecurityConfig.java (cost factor 10 implicito)

```
// INSICURO: BCryptPasswordEncoder() senza argomento usa strength=10,
// valore di default del 2013, insufficiente su hardware moderno.
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // strength=10 implicito
}
```

Codice Vulnerabile — AuthServiceImpl.java (nessun rehash progressivo)

```
// INSICURO: anche alzando il cost factor, gli hash esistenti
// restano a strength=10 finche' l'utente non cambia password.
// Nessun aggiornamento automatico al login.
if (user.getFailedAttempts() > 0) {
    user.setFailedAttempts(0);
    user.setLockedUntil(null);
    userRepository.save(user);
}
// Hash con cost=10 rimane in DB invariato
```

Mitigazione applicata Sono stati applicati due interventi distinti e complementari:

1. **Upgrade cost factor a 12** in `SecurityConfig`: tutti i nuovi hash (registrazione + rehash) vengono calcolati con strength 12, che corrisponde a circa 400 ms su hardware contemporaneo, allineandosi alla raccomandazione OWASP.

2. **Lazy rehash on login**: ad ogni login riuscito, il metodo `passwordEncoder.upgradeEncoding(storedHash)` verifica se l'hash corrente è stato calcolato con un cost factor inferiore a quello configurato. Se sì, la password in chiaro (disponibile solo in questo momento) viene ricalcolata e persistita, migrando silenziosamente l'intero parco utenti senza downtime né password reset forzati.

L'evento di rehash è tracciato con log strutturato `[AUTH] PASSWORD_REHASHED` per audit e monitoraggio della progressione della migrazione.

Codice Sicuro — SecurityConfig.java (cost factor 12)

```
// SICURO: cost factor 12 (OWASP-raccomandato per hardware moderno,
// ~400 ms per hash; valore di default era 10 = ~100 ms).
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12);
}
```

Codice Sicuro — AuthServiceImpl.java (lazy rehash on login)

```java
// SICURO: dopo un login riuscito, verifica se l'hash in DB usa
// un cost factor inferiore a quello corrente (12). Se necessario,
// ricalcola e persiste l'hash aggiornato (lazy migration).
final boolean needsRehash =
    passwordEncoder.upgradeEncoding(user.getPasswordHash());
final boolean needsReset = user.getFailedAttempts() > 0;
if (needsRehash) {
    user.setPasswordHash(passwordEncoder.encode(request.password()));
    log.info("[AUTH] PASSWORD_REHASHED | user={}", user.getUsername());
}
if (needsReset) {
    user.setFailedAttempts(0);
    user.setLockedUntil(null);
}
if (needsRehash || needsReset) {
    userRepository.save(user); // unica save anche se entrambi true
}
```

Test Aggiunti 2 test in `AuthServiceImplTest`: `loginRehashesPasswordWhenEncodingNeedsUpgrade` (verifica che il nuovo hash venga persistito quando `upgradeEncoding` restituisce `true`) e `loginSkipsRehashWhenEncodingIsUpToDate` (verifica che nessuna `save` venga eseguita quando l'hash è già aggiornato e i tentativi falliti sono zero).

Riferimento SEI CERT La mitigazione applica la regola **MSC62-J** del SEI CERT Java Coding Standard: *"Store passwords using a hash function specifically designed for the purpose"*. BCrypt con cost 12 è la scelta corretta per il 2025/26 secondo OWASP; MD5 e SHA-1 (funzioni general-purpose) non sono accettabili.

Riferimento Commit: `fix(security): T-AUTH-03 - BCrypt cost 12 + lazy rehash on login on auth-service`

Upgrade T-AUTH-03 — Argon2id e Password Complexity Policy (esercizi OWASP)

Argon2id — KDF memory-hard L'analisi degli esercizi OWASP Top10 ha evidenziato un ulteriore gap: BCrypt, pur con cost 12, è un algoritmo *memory-weak* — il calcolo di un singolo hash usa al massimo pochi kB di RAM, rendendo gli attacchi con GPU/ASIC efficienti. **Argon2id** (vincitore del Password Hashing Competition 2015, raccomandato da NIST SP 800-63B) è *memory-hard*: il parametro `memory` forza l'attaccante a allocare RAM proporzionale, abbattendo drasticamente il throughput su hardware parallelo. Classificato OWASP A04:2025 — Cryptographic Failures.

1. **DelegatingPasswordEncoder** (default: `argon2`): nuove password codificate come `{argon2}$argon2id$v=19$m=19456,t=2,p=1$...$...` Parametri: memoria 19 MiB, 2 iterazioni, parallelismo 1, coerenti con l'esercizio A04:2025.

2. **Legacy BCrypt**: hash già presenti nel DB mantengono il prefisso `{bcrypt}$2a$12$...`. Il meccanismo di lazy rehash esistente (`upgradeEncoding`) li migra ad Argon2id al primo login successivo, senza downtime né password reset forzati.

3. **Dipendenza**: `org.bouncycastle:bcprov-jdk18on:1.84` (già presente nel BOM per CVE-2026-5598; aggiunta come `implementation` dependency).

Codice Sicuro — SecurityConfig.java (Argon2id + DelegatingPasswordEncoder)

```
// SICURO: Argon2id memory-hard (19 MiB, 2 iter, parallelismo 1)
// DelegatingPasswordEncoder gestisce anche gli hash BCrypt legacy.
@Bean
public PasswordEncoder passwordEncoder() {
    final Map<String, PasswordEncoder> encoders = new HashMap<>();
    encoders.put("argon2", new Argon2PasswordEncoder(
            16, 32, 1, 19 * 1024, 2)); // salt=16B, hash=32B
    encoders.put("bcrypt", new BCryptPasswordEncoder(12));
    return new DelegatingPasswordEncoder("argon2", encoders);
}
```

Password Complexity Policy — A07:2021 L'esercizio A07:2021 richiedeva la validazione della password in fase di registrazione: almeno **16 caratteri**, **2 maiuscole**, **2 cifre**, **2 caratteri speciali**. La baseline accettava password di soli 8 caratteri senza requisiti di complessità (`@Size(min = 8)`), classificato OWASP A07:2021 — Identification and Authentication Failures.

La mitigazione applica un'annotazione `@Pattern` con lookahead regex su tutti i DTO che accettano password in input: `RegisterRequest`, `CreateUserRequest`, `ChangePasswordRequest` (solo `newPassword`). `LoginRequest` non è soggetto alla policy (l'utente non può modificare la password storica a retroattività).

Codice Sicuro — RegisterRequest.java (password complexity)

```
// SICURO: >=16 caratteri, 2 maiuscole, 2 cifre, 2 speciali.
// LoginRequest escluso: la policy non e' retroattiva sul login.
@NotBlank
@Pattern(
    regexp = "^(?=.*[A-Z].*[A-Z])(?=.*[0-9].*[0-9])"
          + "(?=.*[^A-Za-z0-9].*[^A-Za-z0-9]).{16,}$",
    message = "PASSWORD_TOO_WEAK")
String password
```

Riferimento Commit: `feat(security): T-AUTH-03 upgrade - Argon2id KDF + password complexity policy (A04+A07:2025)`

3.1.6 Refresh Token Rotation e Blacklist [T-AUTH-04]

Descrizione La baseline emetteva un unico token JWT con scadenza di **24 ore** privo di un meccanismo di revoca. Un token intercettato (es. via XSS o man-in-the-middle su HTTP) rimaneva valido per l'intera finestra temporale, senza alcuna possibilità di

invalidarlo lato server. L'assenza di *token rotation* aggravava il problema: un refresh token mai revocato poteva essere riutilizzato a tempo indeterminato, anche dopo il logout esplicito dell'utente. Classificato OWASP A07 — Authentication Failures.

Codice Vulnerabile — JwtService.java + AuthController.java (baseline)

```
// Un solo token JWT con TTL 24 ore, nessun refresh token separato
@Value("${jwt.expiration}")
private long jwtExpiration; // 86_400_000 ms = 24 ore

// AuthController: un unico cookie "jwt", nessun cookie "refresh_token"
private static final int COOKIE_MAX_AGE = 86_400; // 24 ore

// Login: emette un solo token
return new AuthResponse(
    jwtService.generateToken(user.getUsername(), user.getRole()));

// Logout: azzera solo il cookie "jwt"
// Il token non viene blacklistato: se qualcuno lo ha catturato,
// rimane valido fino alla sua scadenza naturale (24 ore)
ResponseEntity.ok()
    .header(HttpHeaders.SET_COOKIE, createCookie("", 0).toString())
    .build();
```

Mitigazione applicata Sono stati applicati quattro interventi coordinati:

1. **Separazione access/refresh token**: il login emette ora due JWT distinti: un *access token* con TTL **15 minuti** e un *refresh token* con TTL **7 giorni**. Il refresh token porta una claim `jti` (UUID univoco) e una claim `typ=refresh` che lo distingue dall'access token.

2. **Blacklist Redis**: al momento del refresh o del logout, il JTI del token consumato viene scritto in Redis con chiave `rt:blacklist:<jti>` e TTL pari alla vita residua del token. L'entry si auto-cancella alla scadenza naturale, senza job di pulizia.

3. **Token rotation**: `POST /api/v1/auth/refresh` valida il token ricevuto nel cookie `refresh_token`, ne blacklista il JTI, e risponde con un **nuovo** token pair completo. Ogni refresh produce una coppia fresca — un token intercettato prima della rotation risulta immediatamente blacklistato.

4. **Frontend interceptor**: l'interceptor Axios intercetta i 401, tenta silenziosamente il refresh e riprova la richiesta originale. Nessun logout visibile per sessioni attive entro i 7 giorni; logout automatico se il refresh token è scaduto o revocato.

Codice Sicuro — JwtService.java (generazione refresh token con jti)

```
// Refresh token: TTL 7 giorni, jti UUID, claim typ=refresh
public String generateRefreshToken(String username, Role role) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("role", role.name());
    claims.put("typ", "refresh"); // distingue da access token
    return buildRefreshToken(
        claims, username, refreshExpiration,
```

```java
        UUID.randomUUID().toString()); // jti univoco
}

// Metodi di supporto per la blacklist
public String extractJti(String token) {
    return extractClaim(token, Claims::getId);
}
public Instant extractExpirationInstant(String token) {
    return extractClaim(token, c -> c.getExpiration().toInstant());
}
public boolean isRefreshToken(String token) {
    return "refresh".equals(
        extractClaim(token, c -> c.get("typ", String.class)));
}
```

Codice Sicuro — RefreshTokenServiceImpl.java (blacklist Redis)

```java
// Redis key: "rt:blacklist:<jti>" con TTL = vita residua del token
@Override
public void blacklist(String jti, Instant expiresAt) {
    long ttlSeconds = Duration.between(Instant.now(), expiresAt).getSeconds();
    if (ttlSeconds > 0) {
        // Scadenza automatica: nessun job di pulizia necessario
        redisTemplate.opsForValue()
            .set("rt:blacklist:" + jti, "1", Duration.ofSeconds(ttlSeconds));
    }
}

@Override
public boolean isBlacklisted(String jti) {
    return Boolean.TRUE.equals(redisTemplate.hasKey("rt:blacklist:" + jti));
}
```

Codice Sicuro — AuthServiceImpl.java (refresh con rotation)

```java
@Override
@Transactional(readOnly = true)
public AuthResponse refresh(String refreshToken) {
    // 1. Verifica che sia un refresh token
    if (!jwtService.isRefreshToken(refreshToken)) {
        log.warn("[AUTH] REFRESH_REJECTED | reason=NOT_REFRESH_TOKEN");
        throw new BadCredentialsException("INVALID_REFRESH_TOKEN");
    }
    // 2. Verifica che il JTI non sia in blacklist
    String jti = jwtService.extractJti(refreshToken);
    if (jti == null || refreshTokenService.isBlacklisted(jti)) {
        log.warn("[AUTH] REFRESH_REJECTED | reason=TOKEN_BLACKLISTED");
        throw new BadCredentialsException("INVALID_REFRESH_TOKEN");
    }
    // 3. Verifica che l'utente esista ancora
    String username = jwtService.extractUsername(refreshToken);
    UserAccount user = userRepository.findByUsername(username)
        .orElseThrow(() -> new BadCredentialsException("INVALID_REFRESH_TOKEN"));
```

```java
    // 4. Blacklista il token consumato (rotation)
    refreshTokenService.blacklist(jti, jwtService.extractExpirationInstant(
        ↪ refreshToken));
    // 5. Emette il nuovo token pair
    log.info("[AUTH] REFRESH_SUCCESS | user={}", username);
    return new AuthResponse(
        jwtService.generateToken(username, user.getRole()),
        jwtService.generateRefreshToken(username, user.getRole()));
}
```

Codice Sicuro — api.ts (Axios interceptor con silent refresh)

```java
// Interceptor: su 401, tenta refresh silenzioso prima di fare logout
api.interceptors.response.use(
  (response) => response,
  async (error) => {
    const url = error.config?.url ?? '';
    if (error.response?.status === 401) {
      if (url.includes('/refresh')) {
        performLogout(); return Promise.reject(error); // refresh scaduto
      }
      if (!url.includes('/login') && !url.includes('/me')) {
        if (isRefreshing) {
          // Accoda la richiesta --verra' ritentata dopo il refresh
          return new Promise((resolve, reject) => {
            pendingQueue.push({ resolve, reject });
          }).then(() => api(originalConfig));
        }
        isRefreshing = true;
        try {
          await api.post('/api/v1/auth/refresh'); // cookie gestito dal browser
          drainQueue(true, null);
          return api(originalConfig);        // riprova la richiesta originale
        } catch (err) {
          drainQueue(false, err); performLogout();
        } finally { isRefreshing = false; }
      }
    }
    return Promise.reject(error);
  }
);
```

Test Aggiunti 6 test in `AuthServiceImplTest`: `refreshSuccessIssuesNewTokenPair` (verifica rotation e blacklist del JTI consumato), `refreshThrowsWhenTokenIsNotRefreshType`, `refreshThrowsWhenJtiIsBlacklisted`, `refreshThrowsWhenUserNotFound`, `invalidateRefreshTokenBlacklistsValidToken`, `invalidateRefreshTokenSilentlyHandlesInvalidToken`. Aggiunti 5 test in `RefreshTokenServiceImplTest` che verificano il comportamento della blacklist Redis (store con TTL, skip su token già scaduto, lookup true/false/null).

Effetto La finestra di esposizione di un access token intercettato è ora di **15 minuti** invece di 24 ore. Il refresh token, con TTL 7 giorni, viene ruotato a ogni uso: intercettarlo

è inutile poiché è già blacklistato non appena viene usato dal legittimo titolare. Il logout revoca immediatamente il refresh token, chiudendo la sessione server-side senza dipendere dalla scadenza naturale del token.

Riferimento Commit: `fix(security): T-AUTH-04 -- refresh token rotation + Redis blacklist`

3.1.7 Cambio password non revoca le sessioni attive [T-AUTH-04 residuo]

Descrizione La rotation introdotta con `5ce3010` copre il caso in cui un token viene intercettato durante la trasmissione, ma lasciava aperta una finestra di attacco distinta: un utente che **cambia la propria password** non revocava i refresh token già emessi. Un attaccante in possesso di un refresh token (es. estratto da un dispositivo rubato o da un cookie leak) poteva continuare a ruotarlo per **7 giorni**, ottenendo nuovi access token validi anche dopo che la vittima aveva aggiornato la password. Il sistema non possedeva alcun meccanismo per correlare un token in volo con la "generazione" di credenziali corrente. Classificato OWASP A07 — Authentication Failures.

Codice Vulnerabile — AuthServiceImpl.java (refresh senza controllo versione)

```java
// Prima del fix: refresh accetta qualsiasi token valido non blacklistato,
// indipendentemente da quando e' stato emesso rispetto all'ultimo cambio
    ↪ password.
@Override
@Transactional(readOnly = true)
public AuthResponse refresh(final String refreshToken) {
    // ...
    String username = jwtService.extractUsername(refreshToken);
    UserAccount user = userRepository.findByUsername(username)
        .orElseThrow(() -> new BadCredentialsException("INVALID_REFRESH_TOKEN"));
    // MANCANZA: nessun confronto con la "generazione" corrente delle credenziali
        ↪ .
    // Un token emesso 6 giorni fa (prima del cambio password) supera questo
        ↪ check.
    refreshTokenService.blacklist(jti, jwtService.extractExpirationInstant(
        ↪ refreshToken));
    return new AuthResponse(
        jwtService.generateToken(username, user.getRole(), user.getHotelId()),
        jwtService.generateRefreshToken(username, user.getRole(), user.getHotelId
            ↪ ()));
}

// JwtService.generateToken / generateRefreshToken: nessun claim tv
public String generateToken(String username, Role role, UUID hotelId) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("role", role.name());
    claims.put("hotelId", hotelId.toString());
    // claim "tv" assente: il token non porta informazioni sulla generazione
    return buildToken(claims, username, jwtExpiration);
}
```

Mitigazione applicata L'approccio scelto è il **token versioning**: un contatore monotòno `tokenVersion` in `user_account` viene embedded come claim `tv` in ogni JWT (access e refresh). Al cambio password il contatore viene incrementato e il nuovo valore sovrascrive la chiave Redis `user:tv:<username>`. Il metodo `refresh()` confronta il claim `tv` del token in ingresso con il valore in Redis: se divergono, il token appartiene a una generazione precedente e viene rifiutato con HTTP 401.
Questo approccio è superiore alla blacklist JTI per il caso del cambio password perché:

- Opera come *blanket revocation*: un singolo write Redis revoca *tutti* i token precedenti, senza dover enumerare i JTI.
- Non cresce con il numero di sessioni: la chiave è una sola per utente.
- I token access (TTL 15 min) scadono naturalmente in pochi minuti; la protezione immediata copre i refresh token (TTL 7 giorni).

Migrazione graceful: se la chiave Redis è assente (`storedTv = -1`), il check viene saltato. Questo permette agli utenti che avevano sessioni attive prima del deploy di continuare a usarle. La chiave viene scritta al primo login/register successivo al deploy.

Codice Sicuro — Flyway V4 (migrazione schema)

```sql
-- V4__add_token_version.sql
ALTER TABLE user_account
    ADD COLUMN IF NOT EXISTS token_version INT NOT NULL DEFAULT 0;
COMMENT ON COLUMN user_account.token_version IS
    'Monotonically incrementing counter embedded in JWT "tv" claim. '
    'Incrementing it (on password change) invalidates all previously issued
        ↪ tokens.';
```

Codice Sicuro — JwtService.java (claim tv nei JWT)

```java
private static final String CLAIM_TOKEN_VERSION = "tv";

// generateToken e generateRefreshToken accettano ora tokenVersion come 4
    ↪ argomento
public String generateToken(String username, Role role, UUID hotelId, int
    ↪ tokenVersion) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("role", role.name());
    claims.put("hotelId", hotelId.toString());
    claims.put(CLAIM_TOKEN_VERSION, tokenVersion); // nuovo claim
    return buildToken(claims, username, jwtExpiration);
}

// Estrae il claim tv: -1 se assente (token pre-feature →check saltato)
public int extractTokenVersion(String token) {
    Integer tv = extractClaim(token, c -> c.get(CLAIM_TOKEN_VERSION, Integer.
        ↪ class));
    return tv != null ? tv : -1;
}
```

Codice Sicuro — RefreshTokenServiceImpl.java (Redis token version)

```java
private static final String TV_PREFIX = "user:tv:";

// Scrive "user:tv:<username>" = versione corrente con TTL = lifetime refresh
    ↪ token
@Override
public void storeTokenVersion(String username, int version, Duration ttl) {
    redisTemplate.opsForValue()
        .set(TV_PREFIX + username, String.valueOf(version), ttl);
}

// Restituisce -1 se la chiave non esiste (migrazione graceful)
@Override
public int getTokenVersion(String username) {
    String raw = redisTemplate.opsForValue().get(TV_PREFIX + username);
    if (raw == null) return -1;
    try { return Integer.parseInt(raw); }
    catch (NumberFormatException e) { return -1; }
}
```

Codice Sicuro — AuthServiceImpl.java (check tv in refresh + changePassword)

```java
// In refresh(): check versione prima di ruotare il token
final int storedTv = refreshTokenService.getTokenVersion(username);
final int jwtTv = jwtService.extractTokenVersion(refreshToken);
// storedTv >= 0: chiave presente -> check attivo
// jwtTv = -1 (claim assente) != storedTv positivo -> rifiuto corretto
if (storedTv >= 0 && jwtTv != storedTv) {
    log.warn("[AUTH] REFRESH_REJECTED | reason=TOKEN_VERSION_MISMATCH"
            + " | user={} | jwtTv={} | storedTv={}", username, jwtTv, storedTv);
    throw new BadCredentialsException("INVALID_REFRESH_TOKEN");
}

// POST /api/v1/auth/change-password
@Override
@Transactional
public AuthResponse changePassword(String username, ChangePasswordRequest
    ↪ request) {
    UserAccount user = userRepository.findByUsername(username)
        .orElseThrow(() -> new BadCredentialsException("INVALID_CREDENTIALS"));
    // Secondo fattore: verifica password corrente
    if (!passwordEncoder.matches(request.currentPassword(), user.getPasswordHash
        ↪ ())) {
        log.warn("[AUTH] CHANGE_PASSWORD_REJECTED | reason=BAD_CURRENT_PASSWORD")
            ↪ ;
        throw new BadCredentialsException("INVALID_CURRENT_PASSWORD");
    }
    user.setPasswordHash(passwordEncoder.encode(request.newPassword()));
    user.setTokenVersion(user.getTokenVersion() + 1); // invalida tutte le
        ↪ sessioni
    userRepository.save(user);
    // Sovrascrive Redis: tutti i token con tv=vecchio verranno rifiutati in
        ↪ refresh()
```

```
    refreshTokenService.storeTokenVersion(username, user.getTokenVersion(),
        ↪ REFRESH_TOKEN_TTL);
    log.info("[AUTH] PASSWORD_CHANGED | user={} | newTokenVersion={}", username,
            user.getTokenVersion());
    // Emette un nuovo token pair per mantenere attiva la sessione corrente
    return new AuthResponse(
        jwtService.generateToken(username, user.getRole(), user.getHotelId(),
                                 user.getTokenVersion()),
        jwtService.generateRefreshToken(username, user.getRole(), user.getHotelId
            ↪ (),
                                        user.getTokenVersion()));
}
```

Test Aggiunti 6 test tra `AuthServiceImplTest` e `RefreshTokenServiceImplTest`:

- `refreshThrowsWhenTokenVersionMismatch`: verifica che il refresh venga rifiutato quando il claim `tv` del JWT diverge dal valore Redis (simula un token emesso prima del cambio password).
- `refreshSkipsVersionCheckWhenRedisKeyAbsent`: verifica la migrazione graceful (`storedTv = -1` ⇒ check saltato).
- `changePasswordSuccessIncreasesVersionAndIssuesNewTokenPair`: verifica l'incremento di `tokenVersion`, il write Redis e l'emissione del nuovo token pair.
- `changePasswordThrowsWhenCurrentPasswordIsWrong`: verifica che la password corrente errata blocchi il cambio.
- `changePasswordThrowsWhenUserNotFound`: verifica la gestione di utenti inesistenti.
- `storeTokenVersionWritesCorrectKeyAndValue` e `getTokenVersionReturnsStoredValue` in `RefreshTokenServiceImplTest`.

Effetto Dopo il cambio password, qualsiasi tentativo di ruotare un refresh token "pre-cambio" viene respinto con HTTP 401 non appena il token giunge al `POST /api/v1/auth/refresh`. La finestra di esposizione residua si riduce ai soli access token ancora in volo (massimo **15 minuti**), che scadono naturalmente. Il cambio password reimposta anche i cookie del browser della sessione corrente con il nuovo token pair, garantendo continuità di servizio al legittimo proprietario.

Riferimento Commit: `fix(security): GAP-1 -- password change revokes existing sessions via tokenVersion`

3.1.8 Cambio password obbligatorio non applicato lato server [T-AUTH-06]

Descrizione Trovato durante un giro di navigazione E2E reale con Playwright (non un pentest mirato — `docs/LIVE_E2E_AUDIT_2026-07.md`, BUG-5). Un utente creato da un ADMIN o sottoposto a reset password riceve `mustChangePassword=true`: il frontend reindirizza correttamente a `/settings/password` al login, ma quel reindirizzamento è

l'*unico* controllo esistente. Navigando direttamente a un'altra URL, o chiamando l'API direttamente con le credenziali provvisorie valide, ogni endpoint autenticato rispondeva normalmente: il claim non esisteva nel JWT (solo colonna `must_change_password` in DB, letta esclusivamente da `GET /auth/me`) e nessun filtro server-side lo controllava mai. La rotazione forzata della password era quindi puramente cosmetica. Classificato OWASP A01 (Broken Access Control) / A07 (Identification & Authentication Failures).

Codice Vulnerabile — enforcement assente (baseline)

```
// INSICURO: mustChangePassword non e' nel JWT, nessun filtro lo controlla mai
public String generateToken(String username, Role role, UUID hotelId, int
    ↪ tokenVersion) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("role", role.name());
    claims.put("hotelId", hotelId.toString());
    claims.put("tv", tokenVersion);
    return buildToken(claims, username, jwtExpiration);
    // credenziali provvisorie valide su QUALUNQUE endpoint autenticato
}
```

Mitigazione applicata `mustChangePassword` è ora un claim JWT firmato, aggiunto a `JwtService.generateToken/generateRefreshToken` e popolato su tutti e 4 i percorsi che emettono token: `register`, `login`, `refresh` (propaga lo stato corrente dell'utente) e `changePassword` (emette sempre `false`, coerente col reset appena avvenuto — l'utente non resta bloccato dopo aver adempiuto). L'`AuthenticationFilter` del gateway — unico punto che copre tutte le route business (billing/guest/ frontdesk/fb) e `/api/v1/auth/users/**` — rifiuta con `403 PASSWORD_CHANGE_REQUIRED` qualunque richiesta il cui claim sia `true` e il cui path non sia nell'allow-list (`/api/v1/auth/change-password`, `/me`, `/logout`, `/refresh`); il controllo gira *prima* della RBAC, così nemmeno un ruolo ADMIN può bypassare l'obbligo sfruttando i propri privilegi. Nota architetturale: le route pubbliche di auth-service (login/refresh/change-password/me/logout) non passano comunque da `AuthenticationFilter` (nessun filtro su quella route in `api-gateway.yml`, `permitAll` per design) — l'allow-list è quindi un backstop difensivo, non il gate primario per quei quattro path.

Codice Sicuro — AuthenticationFilter.java (fix)

```
// SICURO: bloccato prima della RBAC, indipendentemente dal ruolo
Boolean mustChangePassword = claims.get("mustChangePassword", Boolean.class);
if (Boolean.TRUE.equals(mustChangePassword)
        && !MUST_CHANGE_PASSWORD_ALLOWED_PATHS.contains(request.getPath().value()
            ↪ )) {
    return this.onError(exchange, "PASSWORD_CHANGE_REQUIRED", HttpStatus.
        ↪ FORBIDDEN);
}
```

Test `AuthServiceImplTest.loginPropagatesMustChangePasswordIntoJwtClaims` verifica che il claim raggiunga `JwtService` con valore `true`; `changePasswordClearsMustChangePasswor` verifica che il token emesso subito dopo il cambio password porti `false` (nessun lockout

residuo). Nuovo gruppo `MustChangePasswordTests` in `AuthenticationFilterTest`: blocco di un endpoint business con `mustChangePassword=true` sia per RECEPTIONIST sia per ADMIN (il ruolo non basta a bypassare), richiesta permessa quando il claim è assente/`false` (retro-compatibilità con token pre-esistenti), e verifica esplicita che un path in allow-list resti raggiungibile nonostante il claim `true`.

Riferimento Commit: `fix(security): T-AUTH-06 -- mustChangePassword enforced server-side via JWT claim`

3.1.9 Riattivazione utente disattivato sempre fallita [T-AUTH-07]

Descrizione Trovato durante un audit E2E reale (`docs/LIVE_E2E_AUDIT_2026-07.md`, BUG-12). Il bottone "Activate" su un utente disattivato in Admin Users falliva **sempre** con 404. `UserManagementServiceImpl.activateUser` cercava il target con `userRepository.findById(targetUserId)`, ma `UserAccount` porta `@SQLRestriction("active = true")` a livello di classe — un filtro Hibernate iniettato in *ogni* query generata per l'entità, comprese le query HQL-derivate ereditate come `findById`, non solo le query JPQL custom. Un utente con `active=false` non veniva quindi mai trovato, e la funzione simmetrica di "Deactivate" — azione di primo livello, non nascosta, presente su ogni riga della tabella — non aveva alcun percorso di ritorno da UI. Il commento nel codice originale ("Must query including inactive — use raw query bypassing @SQLRestriction") descriveva l'intento corretto ma il codice sottostante non lo implementava mai. Classificato OWASP A04 — Insecure Design (il pattern `@SQLRestriction` + query ereditate non è mai stato validato end-to-end per il caso "recupera includendo gli inattivi", un gap di design più che un'iniezione di vulnerabilità).

Codice Vulnerabile — UserManagementServiceImpl.java (baseline)

```java
// INSICURO: findById e' filtrato da @SQLRestriction("active = true")
// su UserAccount -- un utente disattivato non viene MAI trovato
public UserResponse activateUser(UUID hotelId, UUID targetUserId) {
    UserAccount target = userRepository.findById(targetUserId)
        .filter(u -> hotelId.equals(u.getHotelId()))
        .orElseThrow(() -> new NotFoundException("USER_NOT_FOUND")); // sempre
            ↪ qui
    target.setActive(true);
    userRepository.save(target);
    return toResponse(target);
}
```

Mitigazione applicata Nuovo metodo `findByIdAndHotelIdIncludingInactive` in `UserAccountRepository`, basato su query *nativa* anziché JPQL/HQL: a differenza delle query derivate o annotate con HQL, `@SQLRestriction` non viene iniettato in una query nativa scritta a mano — primo caso nel repository, nessun precedente esisteva da copiare. Mantenuto hotel-scoped (predicato `hotel_id` nella query) per non trasformare il bypass in un leak cross-tenant; nome conforme alla regola ArchUnit `TenantIsolationArchTest` (deve contenere `HotelId` nel nome per essere riconosciuto come query tenant-scoped senza necessitare un'esenzione esplicita).

Codice Sicuro — UserAccountRepository.java / UserManagementServiceImpl.java (fix)

```java
// SICURO: query nativa, non soggetta a @SQLRestriction, hotel-scoped
@Query(value = "SELECT * FROM user_account WHERE id = :id AND hotel_id = :
    ↪ hotelId",
       nativeQuery = true)
Optional<UserAccount> findByIdAndHotelIdIncludingInactive(
        @Param("id") UUID id, @Param("hotelId") UUID hotelId);

// Service:
public UserResponse activateUser(UUID hotelId, UUID targetUserId) {
    UserAccount target = userRepository
        .findByIdAndHotelIdIncludingInactive(targetUserId, hotelId)
        .orElseThrow(() -> new NotFoundException("USER_NOT_FOUND"));
    target.setActive(true);
    userRepository.save(target);
    return toResponse(target);
}
```

Test Il test esistente `activateUserShouldSetActiveTrue` mockava `findById` direttamente con Mockito — lo stub bypassava del tutto Hibernate, quindi il test passava anche col bug realmente presente in produzione: nessuna copertura di regressione poteva mai catturarlo. Riscritto per mockare `findByIdAndHotelIdIncludingInactive`; il test `activateUserShouldThrowWhenUserBelongsToAnotherHotel` riscritto per riflettere che ora è la query stessa, hotel-scoped, a non restituire un utente di un altro hotel (non più un filtro applicativo separato). Verificato anche dal vivo via browser reale: utente disattivato → "Activate" → torna "Active" senza alcun intervento diretto sul database.

Riferimento Commit: `fix(security): T-AUTH-07 -- activateUser bypasses @SQLRestriction via native query`

3.1.10 Refresh silenzioso escludeva /me, logout forzato ogni 15 minuti [T-AUTH-08]

Descrizione Trovato durante un audit E2E reale (`docs/LIVE_E2E_AUDIT_2026-07.md`, BUG-8). L'interceptor Axios responsabile del refresh silenzioso (T-AUTH-04) escludeva esplicitamente `/api/v1/auth/me` dal meccanismo di retry-dopo-refresh su un 401, con il commento "Never attempt refresh on auth endpoints themselves to prevent loops". La motivazione è corretta per `/login` (un tentativo di login fallito non deve innescare un refresh), ma sovra-estesa a `/me`: un endpoint di sola lettura, chiamato una sola volta al bootstrap dell'app (`App.tsx`, in un `useEffect` al mount, non ad ogni navigazione client-side). L'access token JWT dura 15 minuti; il refresh token dura 7 giorni ed è pienamente valido. Ma se l'utente ricarica la pagina, apre un link diretto, o la scheda viene ricaricata dal sistema operativo *dopo* che i 15 minuti sono scaduti, `GET /me` falliva con 401, l'interceptor non tentava il refresh, e l'utente veniva reindirizzato a `/login` nonostante un refresh token valido per altri giorni in un cookie httpOnly — un logout forzato che il meccanismo di refresh a 7 giorni esiste esplicitamente per evitare. Classificato

OWASP A07 — Identification & Authentication Failures (nega al legittimo proprietario la persistenza di sessione che il sistema di refresh è progettato per garantire).

Codice Vulnerabile — api.ts (baseline)

```
// INSICURO: /me escluso dal retry-dopo-refresh, nessuna ragione valida
// per un endpoint di sola lettura -- un 401-poi-refresh-poi-retry
// non puo' creare un loop
if (url.includes('/login') || url.includes('/me')) {
  return Promise.reject(error); // niente refresh, logout forzato
}
```

Mitigazione applicata Rimossa l'esclusione di `/me` dalla condizione. Nessun rischio di loop: il refresh stesso viene gestito da un branch dedicato subito dopo (`url.includes('/refresh')`) che chiama `performLogout()` una sola volta e non ritenta mai un secondo refresh; `/me` in fallimento ora percorre lo stesso identico percorso già usato con successo da tutti gli altri endpoint business (refresh silenzioso, poi retry della richiesta originale).

Codice Sicuro — api.ts (fix)

```
// SICURO: solo /login e' escluso -- un login fallito non deve
// mai tentare un refresh, ma /me (sola lettura) si'
if (url.includes('/login')) {
  return Promise.reject(error);
}
if (url.includes('/refresh')) {
  performLogout(); // l'unico punto che chiama logout, mai un retry
  return Promise.reject(error);
}
// ... refresh silenzioso + retry, invariato per /me come per tutti
// gli altri endpoint
```

Test Il test `'rejects immediately on 401 from the /me endpoint without attempting refresh'` codificava il comportamento buggato (nessuna copertura di regressione lo avrebbe mai contraddetto). Riscritto in 2 test: uno verifica che un 401 su `/me` innesca refresh silenzioso e retry (stesso pattern già testato per un 401 generico su un endpoint business), l'altro che un refresh fallito dopo un 401 su `/me` produce comunque un solo logout, non un loop.

Riferimento Commit: `fix(security): T-AUTH-08 -- silent refresh no longer excludes /me`

3.1.11 Cambio password volontario forzava un logout non necessario [T-AUTH-09]

Descrizione Trovato durante un audit E2E reale (`docs/LIVE_E2E_AUDIT_2026-07.md`, BUG-13). `AuthServiceImpl.changePassword` è specificamente progettato per *non* disconnettere l'utente (T-AUTH-04 residuo): incrementa `tokenVersion`, aggiorna la cache

Redis, ed emette un nuovo token pair valido subito dopo — proprio per garantire continuità di sessione dopo un cambio password volontario, comportamento esplicitamente documentato nel codice backend e in questo stesso report. Ma `SettingsPassword.tsx` ignorava questo intento: dopo una risposta `200 OK` (i nuovi cookie erano già stati impostati dal backend), chiamava comunque `logout()`+redirect a `/login` incondizionatamente, buttando via la sessione appena rinnovata. Non è un rischio di sicurezza in senso stretto (il comportamento era per puro accidente "più prudente" del necessario), ma contraddiceva esplicitamente un comportamento backend deliberatamente progettato: l'utente doveva ridigitare da capo la password che aveva appena finito di scrivere due volte nel form. Classificato OWASP A07 — Identification & Authentication Failures (vanifica parzialmente la persistenza di sessione che T-AUTH-04 residuo esiste per garantire).

Codice Vulnerabile — SettingsPassword.tsx (baseline)

```
// INSICURO: il backend ha appena emesso un token pair fresco apposta
// per NON disconnettere l'utente -- questo lo butta via comunque
await authService.changePassword({ currentPassword, newPassword });
addToast(t('password_changed_success'), 'success');
logout();
navigate('/login'); // re-login inutile, l'utente conosce gia' entrambe le
    ↪ password
```

Mitigazione applicata Rimossa la chiamata a `logout()`. Dopo il successo, la sessione viene ri-sincronizzata (`authService.fetchMe()` + `authStore.checkAuth(user)`) così un eventuale `mustChangePassword` stale nello store client (se il cambio veniva dal flusso forzato al primo login) viene azzerato al valore fresco `false` già restituito dal backend, invece di restare appeso finché lo store non viene ricaricato altrove. Navigazione post-successo: alla dashboard se il cambio veniva dal flusso forzato (`mustChangePassword` in arrivo da `location.state`), altrimenti indietro alla pagina di provenienza — mai più a `/login`. Se il refresh della sessione fallisce per una causa indipendente (es. un blip di rete), la navigazione procede comunque: il cambio password stesso è già riuscito, non ha senso bloccare l'utente su quello.

Codice Sicuro — SettingsPassword.tsx (fix)

```
// SICURO: nessun logout, la sessione fresca emessa dal backend resta valida
await authService.changePassword({ currentPassword, newPassword });
addToast(t('password_changed_success'), 'success');
try {
  const freshUser = await authService.fetchMe();
  checkAuth(freshUser); // azzera mustChangePassword stale nello store
} catch {
  // il cambio password e' comunque riuscito, non blocca la navigazione
}
navigate(mustChangePassword ? '/' : -1);
```

Test Il test `'calls changePassword and logs out on success'` codificava il comportamento buggato (asseriva che `logout` venisse chiamato). Riscritto in 3 test: sessione mantenuta viva con 'checkAuth' chiamato con `mustChangePassword: false`, navigazio-

ne alla dashboard quando il cambio veniva dal flusso forzato, e navigazione comunque completata se il refresh post-cambio fallisce.

Riferimento Commit: `fix(security): T-AUTH-09 -- voluntary password change preserves the session`

3.1.12 NullPointerException su login fallito senza clientIp — 401 diventava 500 [T-AUTH-14]

Descrizione Trovato come effetto collaterale durante una sessione di verifica browser end-to-end (2026-09), a seguito di un reset password manuale in ambiente di sviluppo che aveva esposto per caso questo percorso di codice. `AuthServiceImpl.sanitizeForLog(String value)` — il metodo che strippa CR/LF da un valore prima che finisca in un log, per prevenire log forging da username o IP controllati dall'attaccante (CWE-117, T-AUTH-05) — non gestiva `null`: `value.replaceAll("[\r\n]", "_")` lanciava `NullPointerException` se `value` era `null`. `clientIp` è passato a `sanitizeForLog` in tre punti di `login()` (`LOGIN_BLOCKED`, `ACCOUNT_LOCKED`, `LOGIN_FAILED`) e arriva da `@RequestHeader(value = "X-Client-IP", required = false)` in `AuthController` — `null` per qualunque richiesta che raggiunge `auth-service` senza passare dal gateway (in sviluppo, la porta `:8087` è esposta direttamente sull'host, `docker-compose.yml`). Effetto: un login con password errata, o su un account già bloccato, privo di quell'header, restituiva un `500` grezzo invece del normale `401/ACCOUNT_TEMPORARILY_LOCKED` — `loginAttemptService.recordFailure/getLockedUntil` girano comunque prima del logging (il conteggio del lockout non viene bypassato), ma il chiamante riceve un'eccezione non gestita invece della risposta attesa. Classificato OWASP A09 — Security Logging & Monitoring Failures (il meccanismo di audit log introdotto per T-AUTH-05 è il punto di fallimento).

Codice Vulnerabile — AuthServiceImpl.java (baseline)

```java
// INSICURO: nessun controllo null -- clientIp puo' essere null
// (header X-Client-IP opzionale, richiesta diretta senza gateway)
private static String sanitizeForLog(final String value) {
    return value.replaceAll("[\r\n]", "_"); // NPE se value == null
}
```

Mitigazione applicata `sanitizeForLog` reso null-safe: un valore `null` risolve al letterale fisso `"unknown"` invece di propagare l'eccezione. Il letterale non introduce alcun rischio di log forging (nessun contenuto attaccante-controllato) né di information disclosure. Il fix non tocca le chiamate a `loginAttemptService` (che ricevono `clientIp` grezzo, separatamente, per il binding del lockout) né la difesa a tempo costante di T-AUTH-13, entrambe verificate invariate in una revisione di sicurezza dedicata prima del commit.

Codice Sicuro — AuthServiceImpl.java (fix)

```java
// SICURO: null-safe, un login senza clientIp risolve comunque a 401/
    ↪ AccountLocked
private static String sanitizeForLog(final String value) {
```

```
    return value == null ? "unknown" : value.replaceAll("[\r\n]", "_");
}
```

Test Due nuovi test in `AuthServiceImplTest`: `loginWithNullClientIpOnWrongPasswordThrowsCre` (password errata + `clientIp == null` → `BadCredentialsException`, non NPE) e `loginWithNullClientIpOnLockedAccountThrowsAccountLockedNotNpe` (stesso scenario sul branch di lockout → `AccountLockedException`, non NPE).

Follow-up La stessa revisione di sicurezza ha isolato un secondo problema nello stesso percorso di codice, sulla stessa causa (`clientIp` nullo su richiesta diretta ad `auth-service` senza gateway) ma in un punto diverso — la chiave di lockout Redis anziché il logging. Risolto separatamente, vedi [T-AUTH-15] più sotto.

Riferimento Commit: `fix(security): T-AUTH-14 -- sanitizeForLog NPE on null clientIp`

3.1.13 Collasso della chiave di lockout Redis su clientIp nullo [T-AUTH-15]

Descrizione Trovato dalla stessa revisione di sicurezza del fix precedente ([T-AUTH-14]), stessa causa di fondo ma un punto di codice diverso: non il logging, ma la costruzione della chiave Redis del lockout. `LoginAttemptServiceImpl.attemptsKey/lockKey` concatenavano `clientIp` grezzo nella chiave (`login:fail:<username>:<clientIp>`, `login:lock:<username>:<clientIp>`) senza alcun controllo. Quando `clientIp` è `null` — qualunque richiesta che raggiunge `auth-service` senza passare dal gateway, stesso scenario di T-AUTH-14 — la concatenazione Java produce silenziosamente la chiave letterale `...:null`, senza eccezione. Effetto: ogni richiesta priva dell'header `X-Client-IP` per un dato username finisce nello **stesso** bucket di lockout, indipendentemente dalla sorgente reale — un attaccante che raggiunge la porta del servizio direttamente (bypassando il gateway) fa collassare il binding per-`(username, ip)` di T-AUTH-12 in un lockout puramente per-username, riaprendo esattamente il DoS contro il vero proprietario dell'account che T-AUTH-12 era nato per chiudere. Riprodotto dal vivo su un container `auth-service` pre-fix in esecuzione: tre richieste dirette a `:8087` senza header hanno scritto in Redis la chiave letterale `login:fail:admin:null`. Classificato OWASP A07 — Identification & Authentication Failures.

Codice Vulnerabile — LoginAttemptServiceImpl.java (baseline)

```
// INSICURO: nessun controllo su username/clientIp -- un valore null
// produce silenziosamente la chiave letterale "...:null"
private static String attemptsKey(final String username, final String clientIp)
    ↪ {
    return ATTEMPTS_PREFIX + username + ":" + clientIp;
}

private static String lockKey(final String username, final String clientIp) {
    return LOCK_PREFIX + username + ":" + clientIp;
}
```

Mitigazione applicata Fix a due livelli, nessuno dei due sufficiente da solo. Primo livello, `AuthController`: `clientIp` viene risolto prima di chiamare il service — l'header `X-Client-IP` quando presente e non vuoto, altrimenti `HttpServletRequest.getRemoteAddr()` (indirizzo TCP del peer, non un header, quindi non falsificabile dal client per eludere il lockout), stesso fallback già applicato dal rate limiter del gateway sullo stesso header (`RateLimiterConfig.resolveClientIp`). Secondo livello, `LoginAttemptServiceImpl`: username/clientIp nulli o vuoti vengono comunque normalizzati al letterale fisso `"unknown"` prima di costruire la chiave, così il formato `...:null` resta irraggiungibile anche da un futuro chiamante che reintroducesse l'errore.

Codice Sicuro — LoginAttemptServiceImpl.java (fix)

```java
// SICURO: username/clientIp nulli o vuoti risolvono a "unknown",
// mai al letterale ":null"
private static String attemptsKey(final String username, final String clientIp)
    ↪ {
    return ATTEMPTS_PREFIX + normalize(username) + ":" + normalize(clientIp);
}

private static String lockKey(final String username, final String clientIp) {
    return LOCK_PREFIX + normalize(username) + ":" + normalize(clientIp);
}

private static String normalize(final String value) {
    return value == null || value.isBlank() ? UNKNOWN : value;
}
```

Hardening adiacente Trovato nello stesso file durante il fix, stessa radice (nessuna difesa sui dati che tornano da Redis). `getLockedUntil` non proteggeva `Long.parseLong(raw)`: un valore corrotto nella lock key faceva uscire una `NumberFormatException` non gestita da `login()` — un `401` pulito diventava un `500`. Ora fallisce aperto: la chiave corrotta viene cancellata e il metodo restituisce nessun lockout, mentre il contatore dei tentativi (chiave separata) continua comunque a contare. `recordFailure` impostava il TTL del contatore solo al primo tentativo (`attempts == 1`): se quell'`expire()` falliva per un disservizio momentaneo di Redis, il contatore restava senza scadenza e l'account non si sarebbe più sbloccato da solo. Il TTL viene ora riparato ad ogni tentativo in cui risulta assente (`redisTemplate.getExpire`), non solo al primo.

Codice morto rimosso Il meccanismo di lockout pre-Redis (T-AUTH-02) teneva il contatore in Postgres; è interamente superato da Redis ma i residui non erano mai letti né scritti da codice applicativo: colonne `UserAccount.failedAttempts/lockedUntil` e il metodo `UserAccountRepository.updateFailedAttempts` (che portava una `@TenantScopeExempt` — un'eccezione documentata all'isolamento multi-tenant — su un metodo che nessuno chiamava). Migration `V8` droppa le colonne ora inutilizzate; `V2` (già applicata negli ambienti esistenti) resta intatta per non rompere il checksum Flyway.

Test Sei nuovi test in `LoginAttemptServiceImplTest`: `clientIp` nullo o vuoto risolve al bucket `"unknown"`, mai al letterale `":null"`; due IP diversi restano su bucket distinti (non-regressione del binding T-AUTH-12); un valore corrotto nella lock key fallisce aperto e cancella la chiave; un contatore senza TTL viene riparato al tentativo successivo.

Verifica end-to-end Migration Flyway `V1`→`V8` su database pulito e `V8` da sola su una baseline `V1`–`V7` (che simula un ambiente esistente) applicano entrambe senza errori di checksum. Una richiesta diretta al container `auth-service` pre-fix in esecuzione ha riprodotto dal vivo la chiave letterale `login:fail:admin:null` in Redis, confermando la premessa della vulnerabilità sull'ambiente reale di sviluppo.

Riferimento Commit: `fix(security): T-AUTH-15 -- resolve clientIp instead of collapsing lockout keys on null`

3.2 Audit Log e Security Monitoring [Appendice A]

title=Nota metodologica — Implementazioni AI-assisted

Le implementazioni di questa sezione (T-AUTH-05, T-STAY-01, T-STAY-02, T-STAY-03, T-BILL-03) sono state sviluppate con supporto di Claude AI. Il codice è funzionante e integrato nel sistema. La critica critica all'output dell'AI è documentata nell'Appendice B.

3.3 Validazione, Sanificazione ed Encoding

3.3.1 Input Validation su dati ospite [T-GST-02]

Descrizione Il `guest-service` espone endpoint REST per la creazione e l'aggiornamento di profili ospite (`POST /api/v1/guests`, `PUT /api/v1/guests/{id}`). I DTO di richiesta — `GuestRequest` e `IdentityDocumentRequestDTO` — erano protetti solo con `@NotBlank` e `@Size`, ma **privi di vincoli di formato** sui campi testo libero. Un attaccante poteva inviare payload contenenti tag HTML o script nei campi nome, telefono, città e numero documento, oppure una data di nascita nel futuro. Classificato OWASP A05:2025 — Injection.

Codice Vulnerabile — GuestRequest.java (baseline)

```java
public record GuestRequest(
    @NotBlank @Size(max = 100) String firstName, // nessun @Pattern
    @NotBlank @Size(max = 100) String lastName, // nessun @Pattern
    @NotBlank @Email @Size(max = 150) String email,
    @Size(max = 20) String phone,            // nessun @Pattern
    @Size(max = 255) String address,         // nessun @Pattern
    @Size(max = 50) String city,             // nessun @Pattern
    @Size(max = 50) String country,          // nessun @Pattern
    LocalDate dateOfBirth) {                 // nessun @Past
}
// Payload valido inviabile: firstName = "<script>alert(1)</script>"
```

```
// oppure phone = "'; DROP TABLE guests;--"
```

Codice Sicuro — GuestRequest.java + ValidationConstants.java

```
// ValidationConstants.java
public static final String NAME_PATTERN = "^[\\p{L} '\\-]+$";
public static final String PHONE_PATTERN = "^[+]?[0-9 ()\\-]+$";
public static final String TEXT_SAFE_PATTERN = "^[^<>&\"]+$";
public static final String LOCATION_PATTERN = "^[\\p{L}0-9 '\\-.]+$";
public static final String DOCUMENT_NUMBER_PATTERN = "^[A-Za-z0-9\\-]+$";

// GuestRequest.java
public record GuestRequest(
    @NotBlank @Size(max = MAX_FIRST_NAME_LENGTH)
    @Pattern(regexp = NAME_PATTERN) String firstName,

    @NotBlank @Size(max = MAX_LAST_NAME_LENGTH)
    @Pattern(regexp = NAME_PATTERN) String lastName,

    @NotBlank @Email @Size(max = MAX_EMAIL_LENGTH) String email,

    @Size(max = MAX_PHONE_LENGTH)
    @Pattern(regexp = PHONE_PATTERN) String phone,

    @Size(max = MAX_ADDRESS_LENGTH)
    @Pattern(regexp = TEXT_SAFE_PATTERN) String address,

    @Size(max = MAX_LOCATION_LENGTH)
    @Pattern(regexp = LOCATION_PATTERN) String city,

    @Size(max = MAX_LOCATION_LENGTH)
    @Pattern(regexp = LOCATION_PATTERN) String country,

    @Past LocalDate dateOfBirth) {
}

// IdentityDocumentRequestDTO.java
public record IdentityDocumentRequestDTO(
    @NotNull DocumentType documentType,
    @NotBlank @Size(max = MAX_DOCUMENT_NUMBER_LENGTH)
    @Pattern(regexp = DOCUMENT_NUMBER_PATTERN) String documentNumber,
    @NotNull @Past LocalDate issueDate,
    @NotNull @FutureOrPresent LocalDate expiryDate,
    @Size(max = MAX_COUNTRY_LENGTH)
    @Pattern(regexp = LOCATION_PATTERN) String issuingCountry) {
}
```

Effetto Jakarta Bean Validation rifiuta automaticamente al layer controller qualsiasi richiesta che non rispetti i pattern dichiarati: la risposta è `400 Bad Request` prima che il dato raggiunga il service o il database. I pattern scelti coprono nomi internazionali (Unicode `\p{L}`), numeri di telefono in formato internazionale, e bloccano esplicitamente i caratteri usati negli attacchi XSS e SQL Injection (`<`, `>`, `&`, `"`, `;`).

Riferimento SEI CERT Regola **IDS00-J**: *"Sanitize untrusted data passed across a trust boundary"*. Principio chiave: **la normalizzazione deve precedere la validazione**; i pattern allow-list bloccano esplicitamente i caratteri usati in attacchi XSS e SQL Injection (`<`, `>`, `;`, `&`).

Riferimento Commit: `fix(security): T-GST-02 + T-GST-04 -- input validation patterns and explicit mass-assignment guards on guest-service`

3.3.2 Difesa contro Mass Assignment [T-GST-04]

Descrizione Il *mass assignment* è una vulnerabilità per cui un client può impostare campi sensibili dell'entità (chiave primaria, `hotel_id`, flag `active`, timestamp di auditing) includendoli nel payload JSON della richiesta. In Java con Spring e MapStruct il rischio è mitigato dal tipo record (immutabile) e dalla policy `ReportingPolicy.IGNORE`, ma il metodo `GuestMapper#toEntity()` non dichiarava esplicitamente quali target fossero esclusi dal mapping, rendendo la protezione implicita e non verificabile a ispezione statica del codice. Classificato OWASP A06:2025 — Insecure Design.

Codice Vulnerabile — GuestMapper.java (baseline)

```java
@Mapper(componentModel = "spring",
        unmappedTargetPolicy = ReportingPolicy.IGNORE, // protezione implicita
        uses = IdentityDocumentMapper.class)
public interface GuestMapper {

    // Nessun @Mapping(ignore=true): i campi id, hotelId, active,
    // createdAt, updatedAt sono esclusi solo per comportamento di default
    // di ReportingPolicy.IGNORE, non per dichiarazione esplicita.
    Guest toEntity(GuestRequest request);

    @Mapping(target = "id",             ignore = true)
    @Mapping(target = "hotelId",        ignore = true)
    @Mapping(target = "identityDocuments", ignore = true)
    @Mapping(target = "active",         ignore = true)
    @Mapping(target = "createdAt",      ignore = true)
    @Mapping(target = "updatedAt",      ignore = true)
    void updateEntityFromRequest(GuestRequest request, @MappingTarget Guest
        ↪ target);
}
```

Codice Sicuro — GuestMapper.java

```java
@Mapper(componentModel = "spring",
        unmappedTargetPolicy = ReportingPolicy.IGNORE,
        uses = IdentityDocumentMapper.class)
public interface GuestMapper {

    // Ignores espliciti: nessun dato client puo' influenzare questi campi,
    // indipendentemente dalla policy del mapper o da future refactoring.
    // gdprConsentDate aggiunto successivamente nell'ambito della fix T-GST-05
    // (GDPR data retention): campo di sistema impostato dal service, mai dal
        ↪ client.
```

```java
    @Mapping(target = "id",             ignore = true)
    @Mapping(target = "hotelId",        ignore = true)
    @Mapping(target = "identityDocuments", ignore = true)
    @Mapping(target = "active",         ignore = true)
    @Mapping(target = "createdAt",      ignore = true)
    @Mapping(target = "updatedAt",      ignore = true)
    @Mapping(target = "gdprConsentDate", ignore = true)
    Guest toEntity(GuestRequest request);

    @Mapping(target = "id",             ignore = true)
    @Mapping(target = "hotelId",        ignore = true)
    @Mapping(target = "identityDocuments", ignore = true)
    @Mapping(target = "active",         ignore = true)
    @Mapping(target = "createdAt",      ignore = true)
    @Mapping(target = "updatedAt",      ignore = true)
    @Mapping(target = "gdprConsentDate", ignore = true)
    void updateEntityFromRequest(GuestRequest request, @MappingTarget Guest
        ↪ target);
}
```

Effetto La dichiarazione esplicita degli ignore garantisce che qualunque refactoring futuro (rimozione di `ReportingPolicy.IGNORE`, cambio di libreria di mapping) non reintroduca accidentalmente la possibilità di impostare campi sensibili da input client. La protezione diventa *fail-safe by design* anziché dipendente da un comportamento di default.

Riferimento SEI CERT Regola **OBJ13-J**: *"Ensure that references to mutable objects are not exposed"*. La separazione DTO/entità con `@Mapping(target=..., ignore=true)` dichiarativo è la concretizzazione di questo principio: nessun riferimento al modello interno è raggiungibile dall'input client.

Riferimento Commit: `fix(security): T-GST-02 + T-GST-04 -- input validation patterns and explicit mass-assignment guards on guest-service`

3.3.3 Server-side Price Validation [T-FB-02]

Descrizione Il `fb-service` gestisce gli ordini del Food & Beverage (bar, ristorante) addebitati al conto camera. Prima della mitigazione, il DTO `OrderItemRequest` accettava il campo `unitPrice` inviato direttamente dal client. Il service usava quel valore senza alcuna verifica per calcolare il `totalAmount` dell'ordine. Un utente malintenzionato poteva manipolare il payload JSON (e.g. con un proxy interceptor come Burp Suite) e impostare prezzi arbitrari, inclusi valori prossimi allo zero, ottenendo beni e servizi a costo irrisorio senza essere rilevato. Classificato OWASP A06:2025 — Insecure Design.

Codice Vulnerabile — OrderItemRequest.java + service (baseline)

```java
// INSICURO: il prezzo arriva dal client senza validazione
public record OrderItemRequest(
    @NotBlank String itemName,
    @NotNull @Positive Integer quantity,
    @NotNull @Positive BigDecimal unitPrice // attaccante imposta 0.01
```

```
) {}

// INSICURO: totalAmount calcolato con il prezzo del client
BigDecimal totalAmount = order.getItems().stream()
    .map(item -> item.getUnitPrice()     // valore iniettato dal client
                    .multiply(BigDecimal.valueOf(item.getQuantity())))
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Mitigazione applicata L'intervento introduce un catalogo di voci di menu server-side e rimuove il prezzo dal perimetro di controllo del client:

1. **Entita' MenuItem** — nuova tabella `menu_items` (Flyway V2) con campi `name` e `price`. I prezzi sono scritti solo da operatori autenticati con accesso DB; il client non puo' mai influenzarli.
2. **Seed data** — 10 voci tipiche di un hotel (espresso, cappuccino, club sandwich, bistecca...) con prezzi realistici.
3. **OrderItemRequest ridisegnato** — il campo `unitPrice` e `itemName` sono rimossi; il client invia solo `menuItemId` (UUID) e `quantity`.
4. **buildItemsFromCatalog()** — per ogni item richiesto, il service cerca il `MenuItem` per UUID. Se l'ID non esiste lancia `OrderValidationException` (HTTP 400). Il prezzo e' sempre quello del catalogo.
5. **totalAmount ricalcolato server-side** — dopo il lookup, il totale e' la somma di `menuItem.getPrice() * quantity`. Nessun valore del client partecipa al calcolo.

Codice Sicuro — OrderItemRequest.java (fix)

```
// SICURO: nessun campo prezzo esposto al client
public record OrderItemRequest(
    @NotNull UUID menuItemId, // riferimento al catalogo server-side
    @NotNull @Positive Integer quantity
) {}
```

Codice Sicuro — buildItemsFromCatalog() nel service

```
// SICURO: prezzo sempre letto dal catalogo DB, mai dal client
private List<OrderItem> buildItemsFromCatalog(
        List<OrderItemRequest> itemRequests, RestaurantOrder order) {

    List<OrderItem> items = new ArrayList<>();
    for (OrderItemRequest req : itemRequests) {
        MenuItem menuItem = menuItemRepository.findById(req.menuItemId())
            .orElseThrow(() -> new OrderValidationException(
                "MENU_ITEM_NOT_FOUND: " + req.menuItemId()));

        OrderItem item = OrderItem.builder()
            .restaurantOrder(order)
            .itemName(menuItem.getName())
```

```java
            .unitPrice(menuItem.getPrice()) // server-side --mai dal client
            .quantity(req.quantity())
            .build();
        items.add(item);
    }
    return items;
}

// SICURO: totalAmount calcolato sui prezzi del catalogo
BigDecimal totalAmount = items.stream()
    .map(i -> i.getUnitPrice().multiply(BigDecimal.valueOf(i.getQuantity())))
    .reduce(BigDecimal.ZERO, BigDecimal::add);
order.setTotalAmount(totalAmount);
```

V2___add_menu_items.sql — catalogo prezzi server-side

```sql
CREATE TABLE IF NOT EXISTS menu_items (
    id     UUID          NOT NULL DEFAULT gen_random_uuid(),
    name   VARCHAR(255)  NOT NULL,
    price  NUMERIC(10,2) NOT NULL,
    active BOOLEAN       NOT NULL DEFAULT TRUE,
    ...
    CONSTRAINT chk_menu_items_price CHECK (price >= 0)
);

-- Seed: prezzi autoritativi, non modificabili dal client
INSERT INTO menu_items (id, name, price) VALUES
    ('a1000000-...01', 'Espresso',      2.50),
    ('a1000000-...06', 'Club Sandwich', 12.00),
    ('a1000000-...09', 'Bistecca ai Ferri', 22.00);
```

Riferimento Commit: `fix(security): T-FB-02 -- server-side price catalog on fb-service`

3.3.4 Validazione importi pagamento [T-BILL-02]

Descrizione Il `billing-service` accettava importi di pagamento senza vincoli di precisione decimale. Il DTO `PaymentRequest` era vincolato solo con `@Positive` (no negativi, no zero), ma non limitava il numero di cifre decimali. Un attaccante poteva inviare importi con precisione arbitraria (es. `amount: 0.00000000001`) che: (1) bypassano `@Positive` superando il controllo di validazione, (2) inquinano i calcoli di bilancio per arrotondamento cumulativo, (3) possono causare errori di conversione quando il valore viene persistito in colonne `NUMERIC(10,2)` di PostgreSQL. Parallelamente, `transactionReference` non aveva un limite di lunghezza, aprendo a payload di decine di KB con caratteri arbitrari. Classificato OWASP A06:2025 — Insecure Design.

Codice Vulnerabile — PaymentRequest.java (baseline)

```java
// INSICURO: nessun vincolo di precisione decimale
public record PaymentRequest(
```

```
    @NotNull @Positive BigDecimal amount, // accetta 0.00000000001
    @NotNull PaymentMethod paymentMethod,
    @NotBlank String transactionReference) // nessun @Size: payload illimitato
{}

// Il service usa request.amount() raw per tutte le comparazioni:
final BigDecimal balanceDue = invoice.getTotalAmount().subtract(currentTotalPaid
    ↪ );
if (request.amount().compareTo(balanceDue) > 0) { ... }
final BigDecimal newTotalPaid = currentTotalPaid.add(request.amount());
```

Mitigazione applicata L'intervento opera su tre livelli ortogonali:

1. **Precisione decimale** — aggiunto `@Digits(integer = 10, fraction = 2)` su `PaymentRequest.amount` e su `InvoiceRequest.totalAmount`. Jakarta Bean Validation rifiuta con HTTP 400 qualsiasi importo con più di 2 cifre decimali o più di 10 cifre intere, allineato allo standard ISO 4217 per valute a centesimo.

2. **Lunghezza e formato del riferimento transazione** — aggiunto `@Size(max = 100)` e `@Pattern(regexp = "^[A-Za-z0-9\-\_/\.]+$")` su `transactionReference`: limita la lunghezza a 100 caratteri e blocca caratteri di iniezione (`<`, `>`, `;`, `&`, spazi).

3. **Normalizzazione nel service** — difesa in profondità: `PaymentServiceImpl.addPayment()` normalizza l'importo a 2 decimali con `setScale(2, RoundingMode.HALF_UP)` prima di qualsiasi aritmetica e persiste il valore normalizzato tramite `payment.setAmount(paymentAmount)`.

Codice Sicuro — PaymentRequest.java (fix)

```
// SICURO: precisione monetaria imposta a livello DTO
public record PaymentRequest(
    @NotNull(message = "Amount is required")
    @Positive(message = "Amount must be positive")
    @Digits(integer = 10, fraction = 2,
            message = "Amount must have at most 10 integer digits and 2 decimal
                ↪ places")
    BigDecimal amount,

    @NotNull(message = "Payment method is required")
    PaymentMethod paymentMethod,

    @NotBlank(message = "Transaction reference is required")
    @Size(max = 100, message = "Transaction reference must not exceed 100
        ↪ characters")
    @Pattern(regexp = "^[A-Za-z0-9\\-_/\\.]+$",
            message = "Transaction reference contains invalid characters")
    String transactionReference) {}
```

Codice Sicuro — PaymentServiceImpl.java (normalizzazione defense-in-depth)

```java
// SICURO: normalizzazione a 2 decimali prima di ogni aritmetica
final BigDecimal paymentAmount = request.amount()
        .setScale(2, RoundingMode.HALF_UP); // defense-in-depth

if (paymentAmount.compareTo(balanceDue) > 0) {
    log.warn("[BILLING] PAYMENT_REJECTED | reason=PAYMENT_EXCEEDS_BALANCE"
            + " | amount={} | balanceDue={}", paymentAmount, balanceDue);
    throw new BillingValidationException("PAYMENT_EXCEEDS_BALANCE");
}

final Payment payment = paymentMapper.toEntity(request);
payment.setAmount(paymentAmount);         // valore normalizzato persistito
payment.setPaymentDate(LocalDateTime.now());
// ...
final BigDecimal newTotalPaid = currentTotalPaid.add(paymentAmount);
```

Effetto Un importo come `0.00000000001` viene ora rifiutato con HTTP 400 al livello controller prima di raggiungere il service; un eventuale bypass verrebbe comunque normalizzato a `0.00` dalla `setScale` difensiva. Il riferimento transazione non può superare 100 caratteri né contenere caratteri non whitelisted. Due nuovi test coprono il ramo `INVOICE_ALREADY_PAID` e la normalizzazione decimale (`100.005` → `100.01`).

Riferimento Commit: `fix(security): T-BILL-02 -- server-side payment amount validation on billing-service`

3.3.5 Validazione date di prenotazione (range cross-field) [T-RES-03]

Descrizione Il `ReservationRequest` DTO del `reservation-service` validava le date di check-in e check-out singolarmente tramite `@FutureOrPresent`, ma mancava di una validazione *cross-field* che imponesse `checkOutDate > checkInDate`.

Questa lacuna consentiva due classi di input anomali:

1. **Zero-night stay**: `checkIn == checkOut` — la camera appare prenotata, ma la sovrapposizione non viene rilevata perché la condizione `r.checkOutDate > :checkIn` nella query non produce mai un risultato corretto.

2. **Date invertite**: `checkOut < checkIn` — la query di rilevamento overlap (`r.checkInDate < :checkOut AND r.checkOutDate > :checkIn`) fallisce silenziosamente, rendendo la stanza erroneamente disponibile.

Entrambi i casi passano la validazione standard, raggiungono il layer persistenza e corrompono la logica di double-booking prevention introdotta con T-RES-01. La mappatura OWASP è **A03 — Injection** (input non sanificato che altera il comportamento di una query parametrizzata).

Mitigazione applicata **Livello 1 — Bean Validation (controller boundary)**

È stata introdotta l'annotazione `@ValidDateRange`, un vincolo di classe (`@Target(ElementType.TYPE)`) implementato da `DateRangeValidator`:

Codice Vulnerabile — ReservationRequest.java (prima)

```
// Nessuna validazione cross-field: entrambe le date passano
// @FutureOrPresent anche se checkOut <= checkIn
public record ReservationRequest(
    @NotNull @FutureOrPresent LocalDate checkInDate,
    @NotNull @FutureOrPresent LocalDate checkOutDate,
    ...
) { }
```

Codice Sicuro — ReservationRequest.java (dopo)

```
@ValidDateRange // class-level: checkOutDate > checkInDate
public record ReservationRequest(
    @NotNull @FutureOrPresent LocalDate checkInDate,
    @NotNull @FutureOrPresent LocalDate checkOutDate,
    ...
) { }
```

Codice Sicuro — DateRangeValidator.java

```
public class DateRangeValidator
    implements ConstraintValidator<ValidDateRange, ReservationRequest> {

  @Override
  public boolean isValid(ReservationRequest request,
                         ConstraintValidatorContext ctx) {
    if (request == null
        || request.checkInDate() == null
        || request.checkOutDate() == null) {
      return true; // null-check delegato a @NotNull
    }
    return request.checkOutDate().isAfter(request.checkInDate());
  }
}
```

Una violazione produce HTTP 400 con corpo RFC 7807 tramite il `MethodArgumentNotValidException` handler già presente nel `GlobalExceptionHandler`.

Livello 2 — Guard nel service (difesa in profondità)

Per proteggere i chiamanti programmatici che bypassano la validazione del controller, è stato aggiunto un guard statico `verifyDateRange()` invocato come prima operazione in `createReservation()` e `updateReservation()`, prima di qualsiasi accesso a servizi esterni o al DB:

Codice Sicuro — ReservationServiceImpl.java

```
private static void verifyDateRange(ReservationRequest request) {
    if (request.checkInDate() != null
        && request.checkOutDate() != null
        && !request.checkOutDate().isAfter(request.checkInDate())) {
        throw new BadRequestException("CHECKOUT_MUST_BE_AFTER_CHECKIN");
    }
}
```

Test introdotti Sei nuovi test coprono la mitigazione:

- `shouldRejectCreateWhenCheckOutSameDayAsCheckIn` — zero-night stay su create
- `shouldRejectCreateWhenCheckOutBeforeCheckIn` — date invertite su create
- `shouldRejectUpdateWhenCheckOutSameDayAsCheckIn` — zero-night stay su update
- `shouldPassWhenCheckOutIsAfterCheckIn` — caso valido nel validator
- `shouldFailWhenCheckOutSameDayAsCheckIn` — validator diretto
- `shouldFailWhenCheckOutBeforeCheckIn` — validator diretto con range invertito

Riferimento Commit: `fix(security): T-RES-03 -- cross-field date range validation on reservation-service`

3.4 Autorizzazione e Controllo degli Accessi

3.4.1 Strip Header X-Auth dal client [T-GW-01]

*Mitigato nella baseline tramite HMAC-SHA256 (InternalAuthFilter). Documentato nella sezione **Autenticazione Sicura** sopra (T-AUTH-01/T-GW-01/T-FE-02).*

3.4.2 IDOR su risorse ospite/hotel [T-GST-01, T-RES-02, T-BILL-01]

Le vulnerabilità IDOR che consentivano l'accesso cross-tenant a ospiti (T-GST-01), prenotazioni (T-RES-02) e fatture (T-BILL-01) sono state risolte con lo stesso pattern architetturale applicato in cascata a tutti e tre i servizi. Il dettaglio di ciascun intervento è documentato nell'appendice storica (Capitolo 3) alle sezioni dedicate ai rispettivi servizi.

3.4.3 IDOR su ordini F&B — Accesso Cross-Hotel [T-FB-01]

Descrizione Il `fb-service` gestisce gli ordini del ristorante addebitati al conto camera. Nella baseline, l'entità `RestaurantOrder` non possedeva il campo `hotel_id` e tutti i metodi del repository operavano senza scoping multi-tenant: `findAll(pageable)` restituiva *tutti* gli ordini di *tutti* gli hotel, e `findByStayId(stayId)` restituiva gli ordini di qualsiasi hotel purché il `stayId` fosse noto all'attaccante. Un operatore autenticato su Hotel A poteva quindi enumerare e leggere gli ordini F&B dell'Hotel B con una semplice richiesta

`GET /api/v1/fb/orders`. Classificato **OWASP A01 — Broken Access Control / IDOR**.

Codice Vulnerabile — RestaurantOrderRepository.java (baseline)

```java
// INSICURO: nessun filtro hotel_id --tutti gli ordini di tutti gli hotel
@Repository
public interface RestaurantOrderRepository
        extends JpaRepository<RestaurantOrder, UUID> {

    // Restituisce gli ordini di QUALSIASI hotel che abbiano questo stayId
    List<RestaurantOrder> findByStayId(UUID stayId);

    // findAll(pageable) ereditato da JpaRepository: restituisce TUTTI gli ordini
}
```

Codice Vulnerabile — RestaurantOrderServiceImpl.java (baseline)

```java
// getAllOrders: nessun filtro hotel
public Page<RestaurantOrderResponse> getAllOrders(Pageable pageable) {
    return orderRepository.findAll(pageable) // cross-hotel leak
            .map(orderMapper::toResponse);
}

// getOrdersByStayId: nessun check ownership
public List<RestaurantOrderResponse> getOrdersByStayId(UUID stayId) {
    return orderRepository.findByStayId(stayId) // cross-hotel IDOR
            .stream().map(orderMapper::toResponse).toList();
}

// createOrder: hotel_id non impostato sull'entita'
public RestaurantOrderResponse createOrder(RestaurantOrderRequest request) {
    RestaurantOrder order = orderMapper.toEntity(request);
    // hotelId mai valorizzato: ordine non scoped a nessun hotel
    order.setStatus(OrderStatus.BILLED_TO_ROOM);
    ...
}
```

Mitigazione applicata L'intervento segue il medesimo pattern già applicato a `guest-service`,`reservation-service` e `billing-service`:

1. **Flyway V3**: aggiunta colonna `hotel_id UUID NOT NULL` alla tabella `restaurant_orders` con indici compositi `(hotel_id)` e `(hotel_id, stay_id)` per le query più frequenti. La stessa migration corregge anche il CHECK constraint sullo stato (`PENDING/PREPARED/DELIVERED/BILLED_TO_ROOM`), che nella baseline elencava valori obsoleti (`OPEN/IN_PROGRESS/SERVED/CANCELLED`).

2. **InternalAuthFilter**: lettura header `X-Auth-Hotel` iniettato dall'API Gateway; rifiuto con `401` se assente; valore memorizzato in `auth.setDetails(hotelId)` e disponibile in tutto il thread corrente.

3. **FeignHeaderConfig**: propagazione di `X-Auth-Hotel` su tutte le chiamate Feign in uscita (es. verso `stay-service`).
4. **RestaurantOrderServiceImpl**: metodo `resolveHotelId()` estrae `hotel_id` dal `SecurityContext` (mai dal client); tutte le operazioni usano query hotel-scoped; `ORDER_CREATED` loggato con `orderId+stayId+hotelId+total` (prefisso `[FB]`).
5. **RestaurantOrderRepository**: nuovi metodi `findByStayIdAndHotelId` e `findAllByHotelId(Pageable)` al posto delle query non scoped.

Codice Sicuro — InternalAuthFilter.java (lettura X-Auth-Hotel)

```java
// T-FB-01: legge X-Auth-Hotel e lo rifiuta se assente
private static final String HEADER_HOTEL = "X-Auth-Hotel";

final String hotelId = request.getHeader(HEADER_HOTEL);
if (!StringUtils.hasText(hotelId)) {
    LOG.warn("[InternalAuthFilter] Missing X-Auth-Hotel header for user={}",
            username);
    rejectRequest(response, "Missing X-Auth-Hotel header"); // 401
    return;
}

UsernamePasswordAuthenticationToken auth =
    new UsernamePasswordAuthenticationToken(
        username, "", List.of(new SimpleGrantedAuthority("ROLE_" + role)));
auth.setDetails(hotelId); // disponibile via auth.getDetails() nel service
SecurityContextHolder.getContext().setAuthentication(auth);
```

Codice Sicuro — RestaurantOrderServiceImpl.java (hotel-scoped)

```java
// Estrae hotel_id dal SecurityContext; mai accettato dal client (T-FB-01)
private UUID resolveHotelId() {
    final Authentication auth =
        SecurityContextHolder.getContext().getAuthentication();
    final Object details = auth.getDetails();
    if (!(details instanceof String hotelIdStr)
            || !StringUtils.hasText(hotelIdStr)) {
        throw new IllegalStateException("X-Auth-Hotel_MISSING");
    }
    return UUID.fromString(hotelIdStr);
}

// createOrder: hotel_id impostato server-side
public RestaurantOrderResponse createOrder(RestaurantOrderRequest request) {
    final UUID hotelId = resolveHotelId();
    RestaurantOrder order = orderMapper.toEntity(request);
    order.setHotelId(hotelId); // mai dal client
    ...
    log.info("[FB] ORDER_CREATED | orderId={} | stayId={} | hotelId={} | total={}
        ↪ ",
            savedOrder.getId(), savedOrder.getStayId(), hotelId,
            savedOrder.getTotalAmount());
}
```

```java
// getAllOrders: solo ordini dell'hotel autenticato
public Page<RestaurantOrderResponse> getAllOrders(Pageable pageable) {
    final UUID hotelId = resolveHotelId();
    return orderRepository.findAllByHotelId(hotelId, pageable)
                        .map(orderMapper::toResponse);
}

// getOrdersByStayId: IDOR-safe --lista vuota se stayId di altro hotel
public List<RestaurantOrderResponse> getOrdersByStayId(UUID stayId) {
    final UUID hotelId = resolveHotelId();
    return orderRepository.findByStayIdAndHotelId(stayId, hotelId)
                        .stream().map(orderMapper::toResponse).toList();
}
```

Test Aggiunti/aggiornati 8 test in `RestaurantOrderServiceImplTest`: mock del `SecurityContext` con `hotelId` in `@BeforeEach`, `clearContext()` in `@AfterEach`; `shouldSetHotelIdFromSecurityContextOnCreate` (verifica che `order.getHotelId()` == hotelId da SecurityContext), `shouldReturnEmptyListForWrongHotelOnStayLookup` (IDOR scenario — lista vuota per stayId di altro hotel), `shouldReturnOnlyHotelOrdersOnPaginatedListing` (getAllOrders usa `findAllByHotelId`).

Riferimento Commit: `fix(security): T-FB-01 -- IDOR fix, hotel_id scope on fb-service orders`

3.4.4 Double Booking Prevention [T-RES-01]

Descrizione In un sistema di prenotazione alberghiera, il *double booking* si verifica quando due richieste concorrenti riescono a prenotare la stessa camera per date sovrapposte. La condizione di gara (race condition) è del tipo *Time-Of-Check/Time-Of-Use* (TOCTOU): entrambe le transazioni superano il controllo di disponibilità prima che l'altra abbia effettuato la scrittura. Classificato OWASP A06:2025 — Insecure Design.

La baseline non implementava alcun meccanismo di locking:

Codice Vulnerabile — Reservation.java (baseline)

```java
@Entity
@Table(name = "reservations")
public class Reservation {
    // Nessun @Version: aggiornamenti concorrenti alla stessa
    // prenotazione sovrascrivono silenziosamente i dati altrui.
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    // ...
}
```

Codice Vulnerabile — ReservationRepository.java (baseline)

```
// Nessun @Lock: il SELECT sul controllo overlap non acquisisce
// alcun lock a livello DB. Due transazioni simultanee possono
// entrambe passare il check e inserire due prenotazioni
// per la stessa camera nelle stesse date.
List<Reservation> findOverlappingReservationsForNew(
    List<UUID> roomIds,
    LocalDate checkIn,
    LocalDate checkOut
);
```

Mitigazione applicata La mitigazione agisce su due livelli complementari:

- **Optimistic locking JPA** (`@Version`): il campo `version` è incrementato ad ogni `UPDATE`. Se due transazioni leggono la stessa versione e tentano di aggiornare la stessa prenotazione, la seconda riceve `ObjectOptimisticLockingFailureException` che viene tradotta in HTTP 409 Conflict dal `GlobalExceptionHandler`.
- **Pessimistic write lock sul controllo overlap** (`@Lock(PESSIMISTIC_WRITE)`): la query `findOverlappingReservationsForNew` esegue un `SELECT ... FOR UPDATE` sulle prenotazioni esistenti che si sovrappongono alle date richieste. Questo serializza le richieste concorrenti per la stessa camera: la seconda transazione viene messa in attesa finché la prima non fa commit, eliminando la finestra TOCTOU.
- **Formula date-overlap** già corretta: `checkInDate < :checkOut AND checkOutDate > :checkIn`, che copre tutti i casi di sovrapposizione parziale e totale.
- **Flyway migration V4**: aggiunge la colonna `version BIGINT NOT NULL DEFAULT 0` alla tabella `reservations`.

Codice Sicuro — Reservation.java (feature/secure-coding-hardening)

```
@Entity
@Table(name = "reservations")
public class Reservation {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    // Optimistic locking: incrementato ad ogni UPDATE.
    // Conflitto concorrente -> ObjectOptimisticLockingFailureException
    // -> HTTP 409 Conflict.
    @Version
    @Column(nullable = false)
    private Long version;

    // ...
}
```

Codice Sicuro — ReservationRepository.java (feature/secure-coding-hardening)

```java
// SELECT ... FOR UPDATE: serializza le richieste concorrenti
// sulla stessa camera. La seconda transazione attende il commit
// della prima, eliminando la race condition TOCTOU.
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("""
    SELECT r FROM Reservation r
    JOIN r.lineItems li
    WHERE r.active = true
    AND r.status NOT IN (CANCELLED, NO_SHOW)
    AND li.roomId IN :roomIds
    AND li.active = true
    AND r.checkInDate < :checkOut
    AND r.checkOutDate > :checkIn
    """)
List<Reservation> findOverlappingReservationsForNew(
    @Param("roomIds") List<UUID> roomIds,
    @Param("checkIn") LocalDate checkIn,
    @Param("checkOut") LocalDate checkOut
);
```

V4___add_version_to_reservations.sql — Flyway migration

```sql
ALTER TABLE reservations
    ADD COLUMN IF NOT EXISTS version BIGINT NOT NULL DEFAULT 0;
```

GlobalExceptionHandler.java — Handler HTTP 409

```java
@ExceptionHandler(ObjectOptimisticLockingFailureException.class)
public ProblemDetail handleOptimisticLockingFailure(
        ObjectOptimisticLockingFailureException ex) {
    LOG.warn("[OptimisticLock] Concurrent modification: {}", ex.getMessage());
    ProblemDetail pd = ProblemDetail.forStatusAndDetail(
        HttpStatus.CONFLICT, "RESERVATION_CONCURRENT_MODIFICATION");
    pd.setTitle("Conflict");
    pd.setType(URI.create("https://hotelpms.com/errors/conflict"));
    return pd;
}
```

Test Aggiunti 3 test in `ReservationServiceImplTest`: `testCreateReservationOverlapThrowsBadRequest` (verifica che il check di overlap rifiuti la prenotazione con HTTP 400), `testUpdateReservationOverlapThrowsBadRequest` (stesso controllo sull'update), `testSaveThrowsOptimisticLockingFailurePropagates` (verifica che `ObjectOptimisticLockingFailureException` propaghi correttamente al GlobalExceptionHandler per la traduzione in HTTP 409).

Riferimento SEI CERT Regola **VNA02-J**: *"Ensure that compound operations on shared variables are atomic"*. Il controllo di sovrapposizione e l'inserimento della pre-

notazione sono operazioni separate; senza locking il check-then-act non è atomico (race condition TOCTOU). Il `SELECT ... FOR UPDATE` serializza le transazioni concorrenti sulla stessa risorsa.

Nota OWASP A10:2025 Il `GlobalExceptionHandler` centralizzato copre anche **A10:2025 — Mishandling of Exceptional Conditions**, categoria nuova dell'edizione 2025: la `ObjectOptimisticLockingFailureException` viene intercettata e tradotta in HTTP 409 con body RFC 7807 invece di propagarsi come 500 generico. Questo è il pattern opposto al *fail-open* citato nelle slide del corso (es. e-commerce che conferma un ordine dopo timeout del gateway di pagamento).

Riferimento Commit:
`fix(security): T-RES-01 -- optimistic lock + pessimistic write on overlap check`

3.5 Gestione Sicura delle Sessioni

3.5.1 Cookie flags e JWT hardening [T-FE-02] / [T-AUTH-04]

Descrizione Il trasporto del token JWT in un cookie con flag di sicurezza appropriati costituisce la prima barriera contro le due categorie di furto di token più comuni: XSS (lettura del token da JavaScript) e intercettazione passiva su canali non cifrati. Parallelamente, la gestione del ciclo di vita del JWT — scadenza breve, rotazione del refresh token, revoca immediata al logout e al cambio password — determina la finestra di esposizione residua in caso di furto riuscito.

Cookie httpOnly (T-FE-02 — baseline) Fin dalla baseline il server imposta il JWT in un cookie con i tre flag fondamentali:

Cookie JWT — AuthController.java (branch `main`)

```java
// httpOnly: JavaScript non puo' leggere il token (difesa XSS)
// Secure:   il cookie e' trasmesso solo su HTTPS
// SameSite=Strict: il browser non invia il cookie in richieste cross-site
private ResponseCookie createCookie(String value, int maxAge) {
    return ResponseCookie.from("jwt", value)
        .httpOnly(true)
        .secure(true)
        .sameSite("Strict")
        .path("/")
        .maxAge(maxAge)
        .build();
}
```

Il flag `httpOnly` elimina la classe di attacchi in cui uno script XSS (anche iniettato via dipendenza npm compromessa) cerca di estrarre il token con `document.cookie` o `localStorage.getItem`. Il flag `Secure` protegge da intercettazione passiva su reti non cifrate. `SameSite=Strict` impedisce l'invio automatico del cookie su richieste

che originano da siti terzi, costituendo la prima linea di difesa CSRF prima ancora del Double-Submit-Cookie di T-GW-05.

Ciclo di vita del token (T-AUTH-04) Il ridisegno completo del ciclo di vita JWT — access token da 15 minuti, refresh token da 7 giorni con rotazione e blacklist Redis, revoca immediata al logout, e token versioning che invalida tutte le sessioni al cambio password — è documentato in dettaglio nelle sottosezioni precedenti:

- **§ Refresh Token Rotation e Blacklist** (T-AUTH-04): la separazione access/refresh token riduce la finestra di esposizione da 24 ore a **15 minuti**; il logout revoca il refresh token server-side tramite blacklist Redis, chiudendo la sessione indipendentemente dalla scadenza naturale.
- **§ Cambio password non revoca le sessioni attive** (T-AUTH-04 residuo): il meccanismo di token versioning (`tv` claim + Redis key `user:tv:<username>`) garantisce che un token emesso prima del cambio password venga rifiutato in fase di refresh, riducendo la finestra di esposizione residua ai soli access token in volo (max 15 minuti).

Effetto complessivo La combinazione dei tre livelli di protezione — cookie httpOnly, access token a breve scadenza, revoca server-side — rende il furto di token una minaccia gestibile: anche intercettando un access token, un attaccante dispone di una finestra massima di 15 minuti prima che il token scada. Un refresh token intercettato è inutile se il legittimo titolare ne fa uso prima (rotation) o se viene revocato esplicitamente (logout, cambio password).

3.5.2 Protezione CSRF [T-GW-05]

Descrizione Il sistema utilizza cookie `httpOnly` con flag `SameSite=Strict` per trasportare il JWT di sessione. Il flag `SameSite=Strict` è una prima linea di difesa efficace, ma non sufficiente: browser datati, proxy aziendali e bug specifici di implementazione possono violarne il rispetto. In assenza di un meccanismo esplicito di verifica dell'origine della richiesta, un sito malevolo può indurre il browser autenticato dell'utente a inviare richieste mutanti (POST, PUT, PATCH, DELETE) verso il sistema — l'attacco noto come **Cross-Site Request Forgery (CSRF)**.

Nella baseline non era presente alcuna protezione CSRF esplicita: ogni richiesta con un cookie JWT valido veniva accettata dall'API Gateway indipendentemente dall'origine. Classificato OWASP A01 — Broken Access Control.

Pattern Vulnerabile — assenza CSRF token (baseline)

```
// INSICURO: il gateway accetta qualsiasi richiesta con cookie JWT valido,
// indipendentemente dall'origine. Un sito malevolo puo' scatenare:
//   POST https://hotel.example.com/api/v1/guests
//   Cookie: jwt=<token valido> (inviato automaticamente dal browser)
// senza che il sito possa leggere il token -- ma la richiesta viene eseguita.

// api-gateway: nessun filtro CSRF nella baseline
// -> tutte le richieste POST/PUT/PATCH/DELETE passano se il cookie JWT e'
    ↪ valido
```

Mitigazione applicata — Double Submit Cookie L'intervento implementa il pattern **Double Submit Cookie** (OWASP CSRF Prevention Cheat Sheet), che è stateless e si integra naturalmente con l'architettura SPA + JWT:

1. **Emissione del token CSRF** (`auth-service`): all'esito positivo di `/login`, `/register` e `/refresh`, il controller emette un cookie `csrf_token` contenente un UUID casuale, con flag `Secure; SameSite=Strict` ma senza `httpOnly`, in modo che il codice JavaScript della SPA possa leggerlo tramite `document.cookie`.

2. **Invio dell'header** (frontend): un Axios request interceptor legge il cookie `csrf_token` e, per ogni richiesta mutante, lo inietta nell'header `X-CSRF-Token`.

3. **Validazione** (`api-gateway/CsrfFilter`): un `GlobalFilter` con ordine `HIGHEST_PRECEDENCE + 2` confronta il valore dell'header con quello del cookie. Se non corrispondono — o se uno dei due è assente — risponde con HTTP 403. I percorsi pre-autenticazione (`/auth/login`, `/auth/register`) sono esclusi perché il token non è ancora stato emesso.

Perché è sicuro: un sito malevolo può indurre il browser a inviare il cookie (questo è il CSRF stesso), ma la *same-origin policy* del browser impedisce al JavaScript esterno di leggere il valore del cookie `csrf_token`. Senza poter leggere il cookie, il sito malevolo non può costruire l'header `X-CSRF-Token` corrispondente, e la richiesta viene rifiutata dal gateway.

Codice Sicuro — CsrfFilter.java (api-gateway)

```java
@Component
public class CsrfFilter implements GlobalFilter, Ordered {

    static final String CSRF_COOKIE_NAME = "csrf_token";
    static final String CSRF_HEADER_NAME = "X-CSRF-Token";

    private static final Set<HttpMethod> SAFE_METHODS = Set.of(
            HttpMethod.GET, HttpMethod.HEAD, HttpMethod.OPTIONS, HttpMethod.TRACE)
                ↪ ;

    private static final Set<String> EXCLUDED_PATHS = Set.of(
            "/api/v1/auth/login", "/api/v1/auth/register");

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain
        ↪ ) {
        HttpMethod method = exchange.getRequest().getMethod();
        if (method == null || SAFE_METHODS.contains(method)) {
            return chain.filter(exchange); // metodi sicuri: nessun controllo
        }
        String path = exchange.getRequest().getPath().value();
        if (EXCLUDED_PATHS.contains(path)) {
            return chain.filter(exchange); // percorsi pre-auth: esclusi
        }

        HttpCookie csrfCookie = exchange.getRequest()
                .getCookies().getFirst(CSRF_COOKIE_NAME);
        String headerValue = exchange.getRequest()
                .getHeaders().getFirst(CSRF_HEADER_NAME);
```

```java
        if (csrfCookie == null || csrfCookie.getValue().isBlank()
                || headerValue == null || headerValue.isBlank()
                || !csrfCookie.getValue().equals(headerValue)) {
            // Cookie assente/blank o mismatch -> CSRF rilevato -> 403
            exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
            return exchange.getResponse().setComplete();
        }
        return chain.filter(exchange); // token valido: passa
    }

    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE + 2; // dopo SecurityHeadersFilter (+1)
    }
}
```

Codice Sicuro — AuthController.java (auth-service): emissione csrf_token

```java
// Emesso su login/register/refresh, cancellato al logout
private static ResponseCookie createCsrfCookie(String value, int maxAge) {
    return ResponseCookie.from("csrf_token", value)
            .httpOnly(false)       // JS deve poterlo leggere
            .secure(true)
            .path("/")
            .maxAge(maxAge)
            .sameSite("Strict")
            .build();
}

// Nel metodo login():
return ResponseEntity.ok()
    .header(SET_COOKIE, createCookie("jwt", ...).toString())
    .header(SET_COOKIE, createCookie("refresh_token", ...).toString())
    .header(SET_COOKIE,
        createCsrfCookie(UUID.randomUUID().toString(), ACCESS_MAX_AGE).toString()
            ↪ )
    .build();
```

Codice Sicuro — api.ts (frontend): Axios request interceptor

```typescript
// Legge il cookie csrf_token (non-httpOnly) e lo inietta come header
function getCsrfToken(): string | null {
  const match = document.cookie.match(/(?:^|;\s*)csrf_token=([^;]*)/);
  return match ? decodeURIComponent(match[1]) : null;
}

api.interceptors.request.use((config) => {
  const method = config.method?.toUpperCase();
  if (method && ['POST','PUT','PATCH','DELETE'].includes(method)) {
    const token = getCsrfToken();
    if (token) config.headers['X-CSRF-Token'] = token;
  }
```

```
  return config;
});
```

Test Aggiunti 10 test in `CsrfFilterTest`: metodi sicuri (GET, HEAD, OPTIONS) passano senza token; percorsi esclusi (login, register) passano senza token; POST/PUT/PATCH/DELETE senza cookie → 403; senza header → 403; cookie blank → 403; header blank → 403; token mismatch → 403; token valido → 200 (POST, PUT, DELETE, PATCH).

Riferimento Commit: `fix(security): T-GW-05 -- CSRF protection via Double Submit Cookie`

3.6 Configurazioni di Sicurezza (Difesa in Profondità)

3.6.1 Security Headers HTTP [T-GW-03]

Descrizione Nella baseline, l'API Gateway non iniettava alcun header HTTP di sicurezza nelle risposte. Un browser che riceva una risposta priva di questi header è esposto a una serie di vettori di attacco ben documentati:

- Assenza di `Strict-Transport-Security` (HSTS): un attaccante on-path può forzare il downgrade della connessione da HTTPS a HTTP.
- Assenza di `X-Content-Type-Options`: il browser può effettuare *MIME sniffing*, eseguendo come script una risposta dichiarata come `application/json`.
- Assenza di `X-Frame-Options`: la pagina può essere incorporata in un `<iframe>` per attacchi di clickjacking.
- Assenza di `Referrer-Policy`: l'URL completo viene inviato come header `Referer` a terze parti, potenzialmente esponendo path con token o ID.
- Assenza di `Permissions-Policy`: le API del browser (fotocamera, microfono, geolocalizzazione) rimangono accessibili a script di terze parti iniettati.

Classificato OWASP A02:2025 — Security Misconfiguration.

Baseline vulnerabile — nessun header di sicurezza nelle risposte

```
// INSICURO: l'API Gateway non aggiunge alcun header di sicurezza.
// HTTP Response example (baseline):
//   HTTP/1.1 200 OK
//   Content-Type: application/json
//   Transfer-Encoding: chunked
//   < nessun HSTS, X-Frame-Options, CSP, X-Content-Type-Options >
```

Mitigazione applicata È stato aggiunto il componente `SecurityHeadersFilter`, un `GlobalFilter` Spring Cloud Gateway che si inserisce all'inizio della catena (`Ordered.HIGHEST_PRECEDENCE + 1`) e registra una callback `beforeCommit` sulla risposta reattiva. La callback viene eseguita immediatamente prima che gli header HTTP vengano inviati sulla rete, garantendo la presenza degli header su ogni risposta del gateway, indipendentemente dalla rotta o dal microservizio downstream.

SecurityHeadersFilter.java — GlobalFilter (api-gateway)

```java
@Component
public class SecurityHeadersFilter implements GlobalFilter, Ordered {

    static final String VALUE_HSTS = "max-age=31536000; includeSubDomains";
    static final String VALUE_XCTO = "nosniff";
    static final String VALUE_XFO  = "DENY";
    static final String VALUE_REFERRER = "no-referrer";
    static final String VALUE_PERMISSIONS =
        "camera=(), microphone=(), geolocation=()";
    // API responses sono JSON: nessun contenuto da renderizzare
    static final String VALUE_CSP =
        "default-src 'none'; frame-ancestors 'none'";

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
                             GatewayFilterChain chain) {
        exchange.getResponse().beforeCommit(() -> {
            HttpHeaders h = exchange.getResponse().getHeaders();
            h.set(HEADER_HSTS,      VALUE_HSTS);
            h.set(HEADER_XCTO,      VALUE_XCTO);
            h.set(HEADER_XFO,       VALUE_XFO);
            h.set(HEADER_REFERRER,  VALUE_REFERRER);
            h.set(HEADER_PERMISSIONS, VALUE_PERMISSIONS);
            h.set(HEADER_CSP,       VALUE_CSP);
            return Mono.empty();
        });
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() { return Ordered.HIGHEST_PRECEDENCE + 1; }
}
```

Test Aggiunti 7 test in `SecurityHeadersFilterTest`: uno per ciascun header (verifica del valore esatto) e uno per la priorità del filtro. Il test utilizza `MockServerWebExchange` per invocare la callback `beforeCommit` tramite `response.setComplete()`, senza avviare alcun contesto Spring.

Riferimento Commit: `fix(security): T-GW-03 + T-FE-04 -- security headers and CSP`

3.6.2 Content Security Policy [T-FE-04]

Descrizione L'assenza di un header `Content-Security-Policy` nella risposta che serve il frontend React lascia il browser privo di istruzioni su quali sorgenti di contenuto siano legittime. In mancanza di una CSP, uno script iniettato via XSS può caricare risorse da origini arbitrarie, esfiltrare dati o eseguire operazioni autenticate per conto della vittima. Classificato OWASP A02:2025 — Security Misconfiguration.

Problema secondario: la configurazione nginx baseline era definita come stringa *inline* nel `Dockerfile`, rendendo impossibile aggiungere header di risposta senza ricostruire l'immagine con un'istruzione RUN lunga e non manutenibile. Il frontend effettuava inoltre chiamate API all'indirizzo assoluto `http://localhost:8080`, rendendo `connect-src 'self'` inapplicabile (origini diverse per porta).

Dockerfile baseline — nginx inline senza header di sicurezza

```
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
RUN rm /etc/nginx/conf.d/default.conf
# Config inline: nessun header di sicurezza, nessun proxy API
RUN echo "server { listen 80; location / { root /usr/share/nginx/html; \
    try_files $uri $uri/ /index.html; } }" \
    > /etc/nginx/conf.d/default.conf
```

api.ts baseline — URL assoluto, incompatibile con CSP strict

```
// INSICURO per CSP: URL assoluto con porta diversa
// -> connect-src 'self' non copre http://localhost:8080
const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL || 'http://localhost:8080',
  withCredentials: true,
});
```

Mitigazione applicata L'intervento si articola in tre modifiche coordinate:

1. **nginx.conf dedicato** con la CSP completa e tutti gli header di sicurezza, copiato nel container via `COPY nginx.conf /etc/nginx/conf.d/default.conf`.
2. **Proxy nginx per `/api/`**: nginx inoltra tutte le richieste `/api/*` all'API Gateway interno (`http://api-gateway:8080`), rendendo le chiamate API same-origin dal punto di vista del browser e consentendo `connect-src 'self'`.
3. **api.ts**: il `baseURL` è cambiato da `'http://localhost:8080'` a stringa vuota (`''`), così Axios usa URL relativi che il browser risolve sulla stessa origine — compatibili sia con il proxy Vite in sviluppo sia con il proxy nginx in produzione.

nginx.conf — CSP + proxy API + security headers

```
server {
    listen 80;
    root /usr/share/nginx/html;
```

```
    add_header Content-Security-Policy
      "default-src 'self'; script-src 'self';
       style-src 'self' 'unsafe-inline'; img-src 'self' data:;
       connect-src 'self'; frame-ancestors 'none';
       form-action 'self'; base-uri 'self'" always;
    add_header X-Content-Type-Options "nosniff"      always;
    add_header X-Frame-Options     "DENY"            always;
    add_header Referrer-Policy     "no-referrer"     always;

    # Proxy API: same-origin per connect-src 'self'
    location /api/ {
        proxy_pass http://api-gateway:8080;
        proxy_set_header X-Real-IP    $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location / { try_files $uri $uri/ /index.html; }
}
```

api.ts — baseURL relativo, same-origin e CSP-compatibile

```
// SICURO: URL relativo -> stesso host/porta del frontend
// Dev: proxy Vite /api/* -> localhost:8080
// Prod: proxy nginx /api/* -> api-gateway:8080
const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL || '',
  withCredentials: true,
});
```

Nota su `style-src 'unsafe-inline'` TailwindCSS inietta classi di utilità come stili inline al runtime; rimuovere `'unsafe-inline'` richiederebbe la generazione di nonce o hash per ogni stile, incompatibile con il meccanismo di purging di Tailwind. Il rischio residuo è accettabile poiché la CSP blocca comunque `script-src` all'origine stessa, neutralizzando il vettore XSS principale.

Riferimento Commit: `fix(security): T-GW-03 + T-FE-04 -- security headers and CSP`

GAP-10 — L'ereditarietà silenziosa di nginx rompeva ogni download dell'app Gli header di sicurezza T-GW-03/T-FE-04 sopra (`X-Frame-Options`, CSP con `frame-ancestors`) erano dichiarati solo nel server block di `frontend/nginx.conf`. La location `/api/` non definisce un proprio blocco `add_header`, quindi li eredita silenziosamente — lo stesso comportamento di ereditarietà di nginx già documentato nel commento del file per le location `/` e `/assets/`, ma mai esteso a `/api/`. La conseguenza non è solo un gap di hardening: fattura PDF, XML FatturaPA e preventivo PDF si scaricano tutti tramite un `<iframe src="/api/..."></iframe>` nascosto same-origin, e una risposta con `X-Frame-Options: DENY` fa sì che il browser rifiuti di caricarla in quell'iframe — ogni download falliva silenziosamente, senza errore visibile e senza file scaricato.

Scoperto durante un audit dal vivo con click reali sul bottone "Scarica PDF" (non solo apertura di modal o esecuzione dei test unitari, che mockano l'iframe e non attraversano mai nginx reale), innescato da una segnalazione diretta dell'utente. Isolata la causa confrontando i percorsi: `curl` e `fetch()` diretti sullo stesso URL restituivano sempre `200` con `Content-Disposition` corretto — solo il caricamento reale nell'iframe falliva, confermando che il blocco avveniva lato browser sull'header ereditato, non lato server.

frontend/nginx.conf — /api/ eredita gli header del server block (baseline)

```
location /api/ {
    set $upstream     http://api-gateway:8080;
    proxy_pass       $upstream;
    proxy_http_version 1.1;
    proxy_set_header Host            $host;
    proxy_set_header X-Real-IP       $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    # Nessun add_header qui: eredita X-Frame-Options: DENY e
    # frame-ancestors 'none' dal server block, bloccando l'iframe
    # same-origin che il download di PDF/XML usa deliberatamente.
}
```

frontend/nginx.conf — /api/ dichiara i propri header (main)

```
location /api/ {
    set $upstream     http://api-gateway:8080;
    proxy_pass       $upstream;
    proxy_http_version 1.1;
    proxy_set_header Host            $host;
    proxy_set_header X-Real-IP       $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    add_header Content-Security-Policy
        "default-src 'self' blob:; script-src 'self'; style-src 'self' 'unsafe-
            ↪ inline'; img-src 'self' data:; font-src 'self'; connect-src 'self';
            ↪  frame-ancestors 'self'; form-action 'self'; base-uri 'self'"
        always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header Referrer-Policy "no-referrer" always;
    add_header Permissions-Policy "camera=(), microphone=(), geolocation=()"
        ↪ always;
    add_header Strict-Transport-Security "max-age=31536000; includeSubDomains"
        ↪ always;
}
```

`SAMEORIGIN`/`frame-ancestors 'self'` permette l'iframe same-origin che l'app usa deliberatamente per i download, mantenendo bloccato l'embedding da qualunque origine esterna — la minaccia reale contro cui l'header esiste. Verificato dal vivo dopo rebuild e redeploy del container `frontend`: un click reale su "Scarica PDF" produce ora il file sul disco (25149 byte, corrispondenti alla dimensione generata dal backend).

Riferimento Commit: `fix(security): stop inheriting X-Frame-Options: DENY on /api/, it broke every download`

3.6.3 Output Encoding e Protezione XSS [T-FE-01]

Descrizione Il frontend React espone dati recuperati dalle API direttamente nel DOM (nomi ospiti, email, descrizioni). Se un componente usasse `dangerouslySetInnerHTML` o manipolazioni DOM dirette (`innerHTML`/`outerHTML`), un payload XSS memorizzato nel database verrebbe interpretato come HTML dal browser, consentendo l'esecuzione di codice arbitrario nel contesto dell'applicazione. Classificato OWASP A05:2025 — Injection.

L'audit del codebase ha confermato che nessun componente utilizza `dangerouslySetInnerHTML`: React JSX escapa automaticamente ogni espressione renderizzata come contenuto. Tuttavia, in assenza di un guard statico, non vi era nulla che impedisse l'introduzione futura del pattern vulnerabile.

Vettore XSS: dangerouslySetInnerHTML con dato da API

```
// INSICURO: interpreta HTML dal dato API come markup reale
// Un payload "firstName": "<img src=x onerror=alert(document.cookie)>"
// verrebbe eseguito dal browser invece di mostrato come testo

function GuestName({ guest }: { guest: GuestResponseDTO }) {
  return (
    <td
      dangerouslySetInnerHTML={{
        __html: `${guest.firstName} ${guest.lastName}`,
      }}
    />
  );
}
```

Assenza di guard statico — nessuna regola ESLint impediva il pattern

```
// eslint.config.js baseline: nessuna regola su dangerouslySetInnerHTML
rules: {
  ...reactPerf.configs.recommended.rules,
  // (nessun vincolo XSS)
},
```

Mitigazione applicata L'intervento ha operato su due livelli complementari:

1. **Enforcement statico via ESLint (`no-restricted-syntax`)**: tre selettori AST vietano i sink XSS primari in tutta la codebase; la violazione causa un errore di build (non un warning) rendendo impossibile introdurre il pattern senza interrompere la pipeline CI.

2. **Test di regressione XSS**: il componente `Guests` viene renderizzato con un payload `<img src=x onerror=alert(1)>` nel campo `firstName`. Il test verifica che (a)

nessun elemento `<img>` con attributo `onerror` sia presente nel DOM e (b) il payload compaia come testo letterale — comportamento garantito dal JSX escaping di React.

eslint.config.js — no-restricted-syntax vieta i sink XSS

```
// T-FE-01: statically ban XSS sinks
'no-restricted-syntax': [
  'error',
  {
    selector:
      'JSXAttribute[name.name="dangerouslySetInnerHTML"]',
    message:
      '[T-FE-01] dangerouslySetInnerHTML is forbidden ' +
      '-- use React JSX expressions to prevent XSS.',
  },
  {
    selector:
      'AssignmentExpression[left.property.name="innerHTML"]',
    message:
      '[T-FE-01] Direct innerHTML assignment is forbidden ' +
      '-- use React JSX expressions to prevent XSS.',
  },
  {
    selector:
      'AssignmentExpression[left.property.name="outerHTML"]',
    message:
      '[T-FE-01] Direct outerHTML assignment is forbidden ' +
      '-- use React JSX expressions to prevent XSS.',
  },
],
```

Guests.test.tsx — test di regressione XSS (T-FE-01)

```
it('should escape XSS payload in guest fields (T-FE-01)', async () => {
  const xssFirst = '<img src=x onerror=alert(1)>';
  const xssEmail = '"><svg onload=alert(2)>';
  vi.mocked(guestService.getAllGuests).mockResolvedValueOnce([
    { id: '1', firstName: xssFirst, lastName: 'XSS',
      email: xssEmail, phone: '', city: '', country: '' },
  ] as never);

  const { container } = render(<Guests />);

  await waitFor(() => {
    // Nessun elemento iniettato: React non ha interpretato il payload come HTML
    expect(container.querySelector('img[onerror]')).toBeNull();
    expect(container.querySelector('svg[onload]')).toBeNull();
    // Payload presente come testo letterale (HTML-escaped da JSX)
    const cells = container.querySelectorAll('td');
    expect(cells[0]?.textContent).toContain(xssFirst);
  });
});
```

Rendering JSX e output encoding React, per scelta architetturale, tratta ogni espressione JSX `{valore}` come testo puro: i caratteri `<`, `>` e `&` vengono convertiti nelle corrispondenti entità HTML prima dell'inserimento nel DOM virtuale. L'unica eccezione esplicita è `dangerouslySetInnerHTML`, ora vietata staticamente. Di conseguenza, qualunque dato restituito dalle API — anche un payload costruito ad arte — viene visualizzato come stringa letterale senza possibilità di esecuzione.

Riferimento Commit: `fix(security): T-FE-01 -- ESLint no-restricted-syntax bans dangerouslySetInnerHTML/innerHTML; XSS test on Guests`

3.6.4 Route Guard basato sul Ruolo [T-FE-03]

Descrizione In React, nascondere un link di navigazione nel layout non impedisce l'accesso alla route corrispondente: un utente malintenzionato può semplicemente digitare l'URL nel browser e raggiungere la pagina. Nella baseline, la route `/owner-dashboard` era protetta esclusivamente dalla condizione `isOwnerOrAdmin` nel `MainLayout` (UI hiding), che rendeva invisibile la voce di menu per il ruolo `RECEPTIONIST`. Il componente `ProtectedRoute` verificava unicamente lo stato di autenticazione, senza alcun controllo sul ruolo. Classificato OWASP A01 — Broken Access Control.

ProtectedRoute.tsx — baseline: solo verifica autenticazione

```
// INSICURO: nessun controllo sul ruolo utente
export const ProtectedRoute = () => {
  const isAuthenticated = useAuthStore((state) => state.isAuthenticated);

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  return <Outlet />; // qualsiasi utente autenticato accede
};
```

App.tsx — baseline: /owner-dashboard accessibile a tutti gli autenticati

```
// INSICURO: ProtectedRoute verifica solo l'autenticazione
<Route element={<ProtectedRoute />}>
  <Route element={<MainLayout />}>
    {/* ... altre route ... */}
    {/* Chiunque sia autenticato puo' digitare /owner-dashboard */}
    <Route path="/owner-dashboard" element={<OwnerDashboard />} />
  </Route>
</Route>
```

Intervento `ProtectedRoute` è stato esteso con una prop opzionale `allowedRoles?: Role[]`. Se specificata, il componente verifica che `user.role` sia incluso nella allowlist; in caso contrario, redirige verso `/` (home). La route `/owner-dashboard` è ora annidata dentro un secondo `ProtectedRoute` con `allowedRoles={['OWNER', 'ADMIN']}`, garantendo l'enforcement a livello di routing e non solo a livello di UI. L'array viene di-

chiarato come costante di modulo (`OWNER_ADMIN_ROLES`) per rispettare la regola ESLint `react-perf/jsx-no-new-array-as-prop`.

ProtectedRoute.tsx — con prop allowedRoles (T-FE-03)

```
import type { Role } from '../types/auth.types';

interface ProtectedRouteProps {
  allowedRoles?: Role[];
}

export const ProtectedRoute = (
  { allowedRoles }: ProtectedRouteProps = {}
) => {
  const isAuthenticated = useAuthStore((state) => state.isAuthenticated);
  const user = useAuthStore((state) => state.user);

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  if (allowedRoles && (!user || !allowedRoles.includes(user.role))) {
    return <Navigate to="/" replace />; // ruolo non autorizzato
  }

  return <Outlet />;
};
```

App.tsx — /owner-dashboard con doppio guard (T-FE-03)

```
// Costante di modulo: evita new array ad ogni render
const OWNER_ADMIN_ROLES = ['OWNER', 'ADMIN'] as const;

// ...

<Route element={<ProtectedRoute />}>    {/* guard: autenticazione */}
  <Route element={<MainLayout />}>
    {/* ... altre route (accessibili a tutti gli autenticati) ... */}
    <Route element={
      <ProtectedRoute allowedRoles={OWNER_ADMIN_ROLES} /> {/* guard: ruolo */}
    }>
      <Route path="/owner-dashboard" element={<OwnerDashboard />} />
    </Route>
  </Route>
</Route>
```

Test di verifica Quattro nuovi casi in `ProtectedRoute.test.tsx`: ADMIN ammesso, OWNER ammesso, RECEPTIONIST rediretto a `/`, utente `null` con `allowedRoles` impostato rediretto a `/`. Tutti i 144 test rimangono verdi; lint zero warnings; TypeScript clean.

Riferimento Commit: `fix(security): T-FE-03 -- role-based route guard on ProtectedRoute`

Estensione — Pattern "role-gated service call" (T-FE-03b) Una variante del medesimo pattern è stata individuata nel `dashboardService`: la funzione `getDashboardStats()` invocava incondizionatamente l'endpoint `/api/v1/reports/owner` (protetto da `@PreAuthorize("hasAnyRole('OWNER','ADMIN')")`), causando HTTP 403 per i ruoli `RECEPTIONIST` e `MANAGER`. La risposta di errore propagava all'intero Zustand store, rendendo la dashboard inutilizzabile per tutti i ruoli non privilegiati. A differenza della vulnerabilità T-FE-03 (accesso diretto alla route), qui la chiamata API non autorizzata avveniva in modo silenzioso, senza che l'utente potesse accorgersene. Il backend era correttamente protetto e non restituiva dati sensibili, ma il design configurava comunque una violazione OWASP A01. Fix: il parametro `isOwnerOrAdmin: boolean` è propagato da `dashboardStore.fetchStats()` a `Dashboard.tsx`; la chiamata al billing report è omessa se `false`, `pendingRevenue` ritorna `null`. Commit: `fix(dashboard): gate owner report behind role + replace stats with daily KPIs`

3.6.5 Protezione Actuator Endpoints [T-CFG-01]

Descrizione Spring Boot Actuator espone endpoint di diagnostica e monitoraggio (`/actuator/health`, `/actuator/prometheus`, `/actuator/info`) sulla stessa porta dell'API applicativa. Nella baseline, tutti e 9 i microservizi pubblicavano la propria porta applicativa (da `8080` a `8087` e `8888`) con questi endpoint liberamente accessibili. In particolare:

- `/actuator/prometheus` espone metriche interne del processo JVM (thread pool, connection pool del database, latenze HTTP, circuit breaker state) — informazioni preziose per un attaccante in fase di reconnaissance.
- `/actuator/health` con `show-details: always` rivela lo stato dei sotto-sistemi (database UP/DOWN, Redis ping, Zipkin) e i dettagli delle loro configurazioni di connessione.

Classificato OWASP A02:2025 — Security Misconfiguration.

Configurazione Vulnerabile — auth-service.yml (baseline)

```yaml
# INSICURO: actuator sulla stessa porta dell'API (8087), accessibile
# a chiunque raggiunga la porta pubblicata nel docker-compose
management:
  endpoints:
    web:
      exposure:
        include: "prometheus,health,info"
  endpoint:
    health:
      show-details: always # dettagli DB/Redis esposti senza autenticazione
```

docker-compose.yml (baseline) — porta applicativa pubblicata

```yaml
auth-service:
  ports:
    - "8087:8087"  # /actuator/prometheus raggiungibile da qualsiasi client
```

Mitigazione applicata La mitigazione separa il piano di gestione dal piano applicativo tramite la proprietà `management.server.port: 8090`. Tutti e 9 i servizi (incluso il config-service) legano ora il management server sulla porta `8090`, che **non viene mai pubblicata** nel `docker-compose.yml`.

Il risultato è il seguente modello di accesso:

- **Internet / client esterno** → `:808x` (API) — nessuna rotta actuator.
- **Prometheus / healthcheck Docker** → `:8090` (management) — raggiungibile solo dall'interno della rete Docker `hotel_network`.
- `show-details` impostato a `when-authorized`: gli endpoint health restituiscono solo `"status": "UP"` a chiamanti anonimi.

Configurazione Sicura — auth-service.yml (feature/secure-coding-hardening)

```
# SICURO: management server separato su porta interna non pubblicata
management:
  server:
    port: 8090           # porta dedicata, non nel blocco "ports" del compose
  endpoints:
    web:
      exposure:
        include: "prometheus,health,info"
  endpoint:
    health:
      probes:
        enabled: true
      show-details: when-authorized # dettagli visibili solo ad autenticati
```

docker-compose.yml (post-hardening) — porta management non pubblicata

```
auth-service:
  ports:
    - "8087:8087"  # solo API esposta; porta 8090 non in "ports"
  healthcheck:
    test: ["CMD-SHELL",
      "wget -qO- http://localhost:8090/actuator/health/liveness || exit 1"]
```

docker/prometheus/prometheus.yml — scraping su porta interna

```
scrape_configs:
  - job_name: "auth-service"
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["auth-service:8090"] # rete Docker interna, non pubblica
```

La stessa configurazione è applicata simmetricamente a tutti e 9 i servizi (`api-gateway`, `auth-service`, `guest-service`, `inventory-service`, `reservation-service`, `stay-service`, `billing-service`, `fb-service`, `config-service`).

Riferimento Commit:
`fix(security): T-CFG-01 -- move actuator to internal-only management port`

3.6.6 Rate Limiting — Estensione a Tutte le Route e Key Resolver Proxy-Aware [T-GW-02]

Descrizione Il rate limiting baseline presentava due vulnerabilità distinte che lo rendevano aggirabile in produzione:

1. **Copertura parziale**: il filtro `RequestRateLimiter` era configurato *unicamente* sulla route `auth-service` (`/api/v1/auth/**`). Le sei route business (guest, inventory, reservation, stay, billing, fb) erano completamente prive di rate limiting. Un attaccante autenticato — o uno script che sfrutti un account compromesso — poteva inondare qualsiasi endpoint business senza alcun limite, causando un denial-of-service verso gli altri tenant del sistema multi-hotel.

2. **Key resolver non proxy-aware**: il `remoteAddrKeyResolver` baseline leggeva il campo TCP `RemoteAddress` dell'exchange. In un deployment con reverse proxy o load-balancer (nginx, AWS ALB), questo campo contiene l'IP del proxy e *non* quello del client originale: tutti i client condividono quindi un unico bucket comune, rendendo il rate limiting di fatto inutile (un singolo attaccante esaurisce il budget dell'intero sistema).

Classificato OWASP A02:2025 — Security Misconfiguration.

Baseline vulnerabile — rate limiting solo su auth, key resolver non proxy-aware

```
# INSICURO 1: le route business non hanno RequestRateLimiter
- id: guest-service
  uri: "http://guest-service:8083"
  predicates:
    - Path=/api/v1/guests/**
  filters:
    - AuthenticationFilter # nessun rate limiter -> flooding illimitato

# INSICURO 2: key resolver usa TCP RemoteAddress; dietro un proxy
# tutti i client condividono il medesimo bucket (IP del proxy)
@Bean
public KeyResolver remoteAddrKeyResolver() {
    return exchange -> Mono.just(
        Objects.requireNonNull(
            exchange.getRequest().getRemoteAddress(), // IP del proxy, non del
                ↪ client!
            "Remote address must not be null")
            .getAddress().getHostAddress());
}
```

Mitigazione applicata

Key resolver proxy-aware Il `remoteAddrKeyResolver` è stato aggiornato per leggere il header `X-Forwarded-For` quando presente, estraendo il primo IP della lista (il client originale secondo la specifica RFC 7239). Solo in assenza del header si ricade sull'`RemoteAddress` TCP.

È stato inoltre introdotto un secondo bean, `userKeyResolver`, dedicato alle route autenticate. Poiché il filtro `AuthenticationFilter` è dichiarato *prima* di `RequestRateLimiter` nella catena di ogni route, quando il rate limiter viene invocato l'header `X-Auth-User` è già stato iniettato nel request mutato. Il resolver legge tale header e assegna a ciascun utente autenticato un bucket indipendente (chiave `"user:<username>"`), cadendo indietro sull'IP se l'header mancasse. Il risultato è che:

- un account compromesso non può esaurire il quota di altri account (isolamento per-utente);
- le richieste pre-autenticazione (es. endpoint non protetti o fallback) continuano ad essere limitate per IP.

Estensione a tutte le route Il filtro `RequestRateLimiter` è stato aggiunto a tutte e sei le route business, con soglie calibrate sul carico atteso di un operatore alberghiero:

- Route autenticate: `replenishRate: 20`, `burstCapacity: 50` (per-user via `userKeyResolver`).
- Route auth: `replenishRate: 5`, `burstCapacity: 10` (per-IP via `remoteAddrKeyResolver`).

RateLimiterConfig.java — key resolver proxy-aware e per-utente (api-gateway)

```java
@Bean
public KeyResolver remoteAddrKeyResolver() {
    return exchange -> {
        final String forwarded =
            exchange.getRequest().getHeaders().getFirst("X-Forwarded-For");
        if (forwarded != null && !forwarded.isBlank()) {
            return Mono.just(forwarded.split(",")[0].trim()); // client originale
        }
        return Mono.just(
            Objects.requireNonNull(
                exchange.getRequest().getRemoteAddress(),
                "Remote address must not be null").getAddress().getHostAddress());
    };
}

@Bean
public KeyResolver userKeyResolver() {
    return exchange -> {
        final String user =
            exchange.getRequest().getHeaders().getFirst("X-Auth-User");
        if (user != null && !user.isBlank()) {
            return Mono.just("user:" + user); // bucket per-utente
        }
        final String forwarded =
            exchange.getRequest().getHeaders().getFirst("X-Forwarded-For");
```

```
        if (forwarded != null && !forwarded.isBlank()) {
            return Mono.just("ip:" + forwarded.split(",")[0].trim());
        }
        return Mono.just("ip:" + Objects.requireNonNull(
            exchange.getRequest().getRemoteAddress(),
            "Remote address must not be null").getAddress().getHostAddress());
    };
}
```

api-gateway.yml — rate limiting su route business autenticate

```
- id: guest-service
  uri: "${GW_GUEST_SERVICE_URI:http://guest-service:8083}"
  predicates:
    - Path=/api/v1/guests/**
  filters:
    - AuthenticationFilter      # 1. valida JWT, inietta X-Auth-User
    - name: RequestRateLimiter  # 2. usa X-Auth-User per bucket per-utente
      args:
        redis-rate-limiter.replenishRate: 20
        redis-rate-limiter.burstCapacity: 50
        redis-rate-limiter.requestedTokens: 1
        key-resolver: "#{@userKeyResolver}"
# Identica configurazione applicata a: inventory, reservation,
# stay, billing, fb-service
```

Test Aggiunti 6 test in `RateLimiterConfigTest`: `remoteAddrKeyResolver` — XFF presente ⇒ primo IP; XFF assente ⇒ RemoteAddress; XFF blank ⇒ RemoteAddress. `userKeyResolver` — `X-Auth-User` presente ⇒ `"user:<username>"`; assente + XFF presente ⇒ `"ip:<xff>"`; entrambi assenti ⇒ `"ip:<remoteAddr>"`.

Riferimento Commit: `fix(security): T-GW-02 -- rate limiting on all routes with proxy-aware key resolvers`

3.6.7 CORS Misconfiguration — Whitelist Header Esplicita [T-GW-04]

Descrizione La politica CORS (Cross-Origin Resource Sharing) dell'API Gateway consente al browser di inviare richieste autenticate dall'applicazione React (porta 5173 in sviluppo) verso il gateway (porta 8080). Poiché `allowCredentials: true` è obbligatorio (le richieste portano il cookie httpOnly JWT), qualsiasi permissività sugli header diventa rischiosa: un header non necessario accettato in preflight potrebbe essere sfruttato da uno script iniettato per inviare informazioni fuori banda verso l'API.

Nella baseline la configurazione YAML dichiarava:

```
allowedHeaders: "*"
```

Il valore wildcard istruisce il gateway a riflettere *qualsiasi* header nella risposta `Access-Control-Allow-Headers` della preflight OPTIONS, indipendentemente da ciò che il server abbia effettivamente bisogno di leggere. In abbinamento a

`allowCredentials: true`, questo amplia la superficie di attacco: script di terze parti iniettati (XSS) possono includere header personalizzati arbitrari — ad esempio `X-Custom-Attack` — che il browser invierà con le credenziali. Classificato OWASP A02:2025 — Security Misconfiguration.

Un secondo problema minore riguardava l'assenza di `maxAge`: senza questa proprietà, il browser non può mettere in cache le risposte alle preflight OPTIONS, generando una richiesta OPTIONS aggiuntiva prima di *ogni* richiesta mutante. Nella pratica d'uso del gestionale (centinaia di operazioni per sessione) questo moltiplica inutilmente le richieste al gateway.

Configurazione Vulnerabile — api-gateway.yml (baseline, cors globalcors)

```yaml
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '[/**]':
            allowedOrigins:
              "${GW_CORS_ALLOWED_ORIGINS:http://localhost:5173}"
            allowedMethods: [GET, POST, PUT, PATCH, DELETE, OPTIONS]
            allowedHeaders: "*" # wildcard: qualsiasi header e' riflesso
            allowCredentials: true
            # maxAge assente: nessuna cache delle preflight
```

Mitigazione applicata La whitelist degli header accettati è stata sostituita con una lista esplicita e minimale:

- **`Content-Type`**: necessario per il body JSON delle richieste POST/PUT/PATCH.
- **`X-CSRF-Token`**: necessario per il meccanismo di protezione Double-Submit-Cookie introdotto in T-GW-05; senza questa voce, il browser non otterrebbe l'approvazione della preflight per l'header e le richieste mutanti fallirebbero in produzione.

L'header `Authorization` non è necessario: il token JWT transita nel cookie httpOnly, mai in un header esplicito.

È stato aggiunto `maxAge: 3600` per istruire il browser a memorizzare in cache il risultato della preflight OPTIONS per un'ora, riducendo il numero di richieste ridondanti senza abbassare la sicurezza.

L'origine consentita era già correttamente vincolata tramite la variabile d'ambiente `GW_CORS_ALLOWED_ORIGINS` (default `http://localhost:5173`), senza l'uso di wildcard né pattern globbing. In produzione, nginx serve il frontend sulla stessa origine del proxy API, rendendo il CORS non necessario in quel contesto.

Configurazione Sicura — api-gateway.yml (feature/secure-coding-hardening)

```yaml
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
```

```
        '[/**]':
          allowedOrigins:
            "${GW_CORS_ALLOWED_ORIGINS:http://localhost:5173}"
          allowedMethods: [GET, POST, PUT, PATCH, DELETE, OPTIONS]
          allowedHeaders:    # whitelist minimale e dichiarativa
            - Content-Type   # body JSON richieste mutanti
            - X-CSRF-Token   # Double-Submit-Cookie (T-GW-05)
          allowCredentials: true
          maxAge: 3600       # cache preflight: 1 ora
```

Effetto Un client cross-origin può includere nei propri header *solo* `Content-Type` e `X-CSRF-Token`; qualsiasi altro header personalizzato viene rifiutato già in fase di preflight dal browser, prima che la richiesta raggiunga il gateway. La combinazione con `allowedOrigins` esplicito e `allowCredentials: true` forma un contratto CORS minimale e corretto: accetta solo ciò che è strettamente necessario al funzionamento dell'applicazione.

Nota sull'assenza di test unitari La configurazione CORS è gestita interamente da Spring Cloud Gateway tramite il blocco YAML `globalcors`, senza alcun bean Java da testare in isolamento. Il comportamento corretto è verificato dall'integrazione end-to-end (Playwright E2E): il browser esegue le preflight verso `http://localhost:5173` e, se la risposta OPTIONS è errata, l'intera suite E2E fallisce a livello di `NetworkError`.

Riferimento Commit: `fix(security): T-GW-04 -- restrict CORS allowedHeaders whitelist and add preflight maxAge`

3.7 Broken Access Control — Isolamento Multi-Tenant

3.7.1 IDOR e Data Leak sul Guest Service [T-GST-01] [T-GST-03]

Descrizione Il sistema gestisce strutture ricettive multiple (multi-hotel). Ogni utente autenticato appartiene a un hotel specifico, identificato dal campo `hotel_id` estratto dal JWT e iniettato dall'API Gateway come header `X-Auth-Hotel`.

T-GST-01 (IDOR): il `GuestController` esponeva endpoint `GET/api/v1/guests/{id}` e `PUT/api/v1/guests/{id}` senza verificare l'ownership dell'hotel. Un receptionist dell'Hotel A, conoscendo o indovinando un UUID, poteva leggere e modificare profili ospite dell'Hotel B. Classificato OWASP A01 — Broken Access Control.

T-GST-03 (Multi-tenant leak): le query `findAll()` e `searchByKeyword()` non filtravano per `hotel_id`, restituendo ospiti di tutti gli hotel alla prima chiamata paginata. I dati PII (nome, documento, data di nascita) risultavano accessibili cross-hotel. Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — GuestServiceImpl.java (baseline)

```java
// INSICURO: lookup per solo UUID, nessun check ownership
private Guest resolveGuest(UUID id) {
    return guestRepository.findById(id) // trova chiunque, qualsiasi hotel
        .orElseThrow(() -> new NotFoundException("GUEST_NOT_FOUND"));
}

// INSICURO: lista tutti gli ospiti di tutti gli hotel
public Page<GuestResponse> getAllGuests(Pageable pageable) {
    return guestRepository.findAll(pageable) // nessun filtro hotel_id
        .map(guestMapper::toResponse);
}
```

Mitigazione applicata L'intervento si articola in quattro livelli:

1. **Schema DB** — Flyway V2 aggiunge la colonna `hotel_id UUID NOT NULL` alla tabella `guests`, con indice parziale su `active = TRUE`.
2. **Extraction dal SecurityContext** — `InternalAuthFilter` legge `X-Auth-Hotel` e lo memorizza in `Authentication.details`. Il service lo estrae con `extractHotelId()` senza dipendere dal controller.
3. **Query scoped** — tutte le query del repository usano varianti hotel-scoped (`findByIdAndHotelId`, `findAllByHotelId`, `searchByKeywordAndHotelId`, `findAllByIdInAndHotelId`).
4. **IDOR-safe 404** — `resolveGuest(id, hotelId)` chiama `findByIdAndHotelId`: se il guest esiste ma appartiene a un altro hotel il metodo restituisce `Optional.empty()` e il service lancia `NotFoundException` (HTTP 404), senza rivelare l'esistenza della risorsa.

Codice Sicuro — GuestServiceImpl.java (fix)

```java
// SICURO: lookup combinato id + hotel_id, IDOR-safe con 404
private Guest resolveGuest(UUID id, UUID hotelId) {
    return guestRepository.findByIdAndHotelId(id, hotelId)
        .orElseThrow(() -> new NotFoundException("GUEST_NOT_FOUND"));
    // 404 anche se il guest esiste ma appartiene ad altro hotel:
    // l'attaccante non puo' distinguere "non esiste" da "non tuo"
}

// SICURO: elenco filtrato per hotel del caller
public Page<GuestResponse> getAllGuests(Pageable pageable) {
    UUID hotelId = extractHotelId();      // da Authentication.details
    return guestRepository
        .findAllByHotelId(hotelId, pageable) // solo il proprio hotel
        .map(guestMapper::toResponse);
}

// SICURO: hotelId estratto dal SecurityContext (non dal client)
private UUID extractHotelId() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    Object details = auth.getDetails();   // iniettato da InternalAuthFilter
```

```
    return UUID.fromString((String) details); // mai controllabile dal client
}
```

InternalAuthFilter.java — estrazione X-Auth-Hotel

```
// Il gateway inietta X-Auth-Hotel dal JWT; il filter lo memorizza nei details
String hotelId = request.getHeader("X-Auth-Hotel");
// Richiesta rifiutata (401) se l'header e' assente
if (!StringUtils.hasText(hotelId)) {
    rejectRequest(response, "Missing required gateway authentication headers");
    return;
}
UsernamePasswordAuthenticationToken auth = new
    ↪ UsernamePasswordAuthenticationToken(
    username, "", List.of(new SimpleGrantedAuthority("ROLE_" + role)));
auth.setDetails(hotelId); // tenant context propagato al service layer
SecurityContextHolder.getContext().setAuthentication(auth);
```

V2___add_hotel_id_to_guests.sql — Flyway migration

```
ALTER TABLE guests
    ADD COLUMN hotel_id UUID NOT NULL
    DEFAULT '00000000-0000-0000-0000-000000000000';
ALTER TABLE guests ALTER COLUMN hotel_id DROP DEFAULT;

CREATE INDEX IF NOT EXISTS idx_guests_hotel_id
    ON guests (hotel_id) WHERE active = TRUE;
```

Riferimento Commit:
`fix(security): T-GST-01 + T-GST-03 -- hotel_id isolation on guest-service`

3.7.2 IDOR sul Reservation Service [T-RES-02]

Descrizione Il `reservation-service` gestisce le prenotazioni di camera. Prima della mitigazione, l'endpoint `GET /api/v1/reservations/{id}` e le operazioni correlate (PUT, DELETE, PATCH) non verificavano che la prenotazione richiesta appartenesse all'hotel del caller. Un receptionist dell'Hotel A, conoscendo o indovinando un UUID valido, poteva leggere, modificare o cancellare prenotazioni dell'Hotel B. Classificato OWASP A01 — Broken Access Control.

Analogamente, `getAllReservations()` eseguiva `findAll()` senza alcun filtro su `hotel_id`, esponendo l'intero dataset di prenotazioni a qualsiasi utente autenticato.

Codice Vulnerabile — ReservationServiceImpl.java (baseline)

```
// INSICURO: lookup per solo UUID, nessun controllo ownership hotel
private Reservation findReservationByIdOrThrow(UUID id) {
    return reservationRepository.findById(id) // restituisce qualsiasi hotel
        .orElseThrow(() -> new NotFoundException("RESERVATION_NOT_FOUND"));
}
```

```java
// INSICURO: lista tutte le prenotazioni di tutti gli hotel
public Page<ReservationResponse> getAllReservations(Pageable pageable) {
    return reservationRepository.findAll(pageable) // nessun filtro hotel_id
        .map(...);
}
```

Mitigazione applicata L'intervento replica il pattern gia' applicato al `guest-service` (T-GST-01/T-GST-03) e si articola in tre livelli:

1. **InternalAuthFilter aggiornato** — il filter legge l'header `X-Auth-Hotel` iniettato dal gateway e lo salva in `Authentication.details`. Richieste prive dell'header sono rifiutate con HTTP 401.

2. **Query hotel-scoped** — il repository espone due nuovi metodi: `findByIdAndHotelId(id,hotelId)` e `findAllByHotelId(hotelId,pageable)`. Il service usa `findByIdAndHotelId` per tutte le operazioni su singola risorsa (GET, PUT, DELETE, PATCH).

3. **IDOR-safe 404** — se la prenotazione esiste ma appartiene a un altro hotel, `findByIdAndHotelId` restituisce `Optional.empty()` e il service lancia `NotFoundException` (HTTP 404), senza rivelare l'esistenza della risorsa all'attaccante.

4. **hotelId impostato alla creazione** — `createReservation` recupera il `hotelId` dal SecurityContext tramite `resolveHotelId()` e lo persiste sull'entità, garantendo che ogni nuova prenotazione sia legata all'hotel del caller.

Codice Sicuro — ReservationServiceImpl.java (fix)

```java
// SICURO: lookup combinato id + hotel_id (IDOR-safe con 404)
private Reservation findReservationByIdAndHotelOrThrow(UUID id, UUID hotelId) {
    return reservationRepository.findByIdAndHotelId(id, hotelId)
        .orElseThrow(() -> new NotFoundException("RESERVATION_NOT_FOUND"));
    // 404 anche se esiste ma appartiene ad altro hotel
}

// SICURO: lista filtrata per hotel del caller
public Page<ReservationResponse> getAllReservations(Pageable pageable) {
    UUID hotelId = resolveHotelId();
    return reservationRepository
        .findAllByHotelId(hotelId, safePageable)
        .map(...);
}

// SICURO: hotelId estratto dal SecurityContext, mai dal client
private UUID resolveHotelId() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    if (auth == null || !(auth.getDetails() instanceof String hotelIdStr)) {
        throw new IllegalStateException("Hotel ID cannot be null");
    }
    return UUID.fromString(hotelIdStr);
```

```
}

// SICURO: hotelId persistito sul record alla creazione
public ReservationResponse createReservation(ReservationRequest request) {
    UUID hotelId = resolveHotelId();
    Reservation reservation = reservationMapper.toEntity(request);
    reservation.setHotelId(hotelId); // tenant-bound, non alterabile dal client
    ...
}
```

ReservationRepository.java — metodi hotel-scoped

```
// Lookup singola risorsa con check ownership (Spring Data query derivata)
Optional<Reservation> findByIdAndHotelId(UUID id, UUID hotelId);

// Lista paginata filtrata per hotel
Page<Reservation> findAllByHotelId(UUID hotelId, Pageable pageable);
```

Riferimento Commit: `fix(security): T-RES-02 -- hotel_id isolation on reservation-service`

3.7.3 IDOR sul Billing Service [T-BILL-01]

Descrizione Il `billing-service` gestisce fatture e pagamenti. Prima della mitigazione, `InvoiceServiceImpl.getInvoice(id)` e `PaymentServiceImpl.addPayment(invoiceId, ...)` usavano `findById(id)`, che restituisce qualsiasi fattura indipendentemente dall'hotel di appartenenza. Un receptionist dell'Hotel A, indovinando un UUID valido, poteva:

- leggere le fatture dell'Hotel B (`GET /api/v1/invoices/{id}`);
- registrare pagamenti su fatture di altri hotel (`POST /api/v1/invoices/{invoiceId}/payments`);
- ottenere la lista completa di tutte le fatture di tutti gli hotel via `getAllInvoices()`.

Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — InvoiceServiceImpl.java (baseline)

```
// INSICURO: lookup per solo UUID, nessun controllo ownership hotel
public InvoiceResponse getInvoice(UUID id) {
    Invoice invoice = invoiceRepository.findById(id) // qualsiasi hotel
        .orElseThrow(() -> new NotFoundException("INVOICE_NOT_FOUND"));
    return invoiceMapper.toResponse(invoice);
}

// INSICURO: lista tutte le fatture di tutti gli hotel
public Page<InvoiceResponse> getAllInvoices(Pageable pageable) {
    return invoiceRepository.findAll(pageable) // nessun filtro hotel_id
        .map(invoiceMapper::toResponse);
}
```

Codice Vulnerabile — PaymentServiceImpl.java (baseline)

```java
// INSICURO: addPayment carica la fattura senza verifica hotel
public PaymentResponse addPayment(UUID invoiceId, PaymentRequest request) {
    Invoice invoice = invoiceRepository.findById(invoiceId)
        .orElseThrow(() -> new NotFoundException("INVOICE_NOT_FOUND"));
    // ... aggiunge il pagamento indipendentemente dall'hotel
}
```

Mitigazione applicata L'intervento replica il pattern consolidato di T-GST-01 e T-RES-02:

1. **InternalAuthFilter aggiornato** — il filter legge l'header `X-Auth-Hotel` iniettato dal gateway e lo salva in `Authentication.details`. Richieste prive dell'header sono rifiutate con HTTP 401.
2. **Query hotel-scoped nel repository** — aggiunti tre metodi: `findByIdAndHotelId(id, hotelId)`, `findByHotelId(hotelId, pageable)` (paginata) e `findFirstByReservationIdAndHotelIdOrderByIssueDateDesc(reservationId, hotelId)`.
3. **IDOR-safe 404** — `findByIdAndHotelId` restituisce `Optional.empty()` se la fattura esiste ma appartiene ad un altro hotel; il service lancia `NotFoundException` (HTTP 404), senza rivelare l'esistenza della risorsa all'attaccante.
4. **hotelId impostato alla creazione** — `createInvoice` imposta `invoice.setHotelId(resolveHotelId())`, ignorando qualsiasi valore passato nel corpo della richiesta.
5. **FeignHeaderConfig aggiornato** — propaga `X-Auth-Hotel` su tutte le chiamate Feign verso `guest-service` e `reservation-service`.

Codice Sicuro — InvoiceServiceImpl.java (fix)

```java
// SICURO: lookup combinato id + hotel_id (IDOR-safe con 404)
public InvoiceResponse getInvoice(UUID id) {
    UUID hotelId = resolveHotelId();
    Invoice invoice = invoiceRepository.findByIdAndHotelId(id, hotelId)
        .orElseThrow(() -> new NotFoundException("INVOICE_NOT_FOUND"));
    return invoiceMapper.toResponse(invoice);
}

// SICURO: lista paginata filtrata per hotel del caller
public Page<InvoiceResponse> getAllInvoices(Pageable pageable) {
    UUID hotelId = resolveHotelId();
    return invoiceRepository.findByHotelId(hotelId, safePageable)
        .map(invoiceMapper::toResponse);
}
```

```java
// SICURO: hotelId estratto dal SecurityContext, mai dal client
public InvoiceResponse createInvoice(InvoiceRequest request) {
    // ... validazioni guest e reservation ...
    Invoice invoice = invoiceMapper.toEntity(request);
    invoice.setHotelId(resolveHotelId()); // tenant-bound
    return invoiceMapper.toResponse(invoiceRepository.save(invoice));
}

private UUID resolveHotelId() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    if (auth == null || !(auth.getDetails() instanceof String hotelIdStr)) {
        throw new IllegalStateException("MISSING_HOTEL_CONTEXT");
    }
    return UUID.fromString(hotelIdStr);
}
```

Codice Sicuro — PaymentServiceImpl.java (fix)

```java
// SICURO: lookup combinato id + hotel_id (IDOR-safe con 404)
public PaymentResponse addPayment(UUID invoiceId, PaymentRequest request) {
    UUID hotelId = resolveHotelId();
    Invoice invoice = invoiceRepository.findByIdAndHotelId(invoiceId, hotelId)
        .orElseThrow(() -> new NotFoundException("INVOICE_NOT_FOUND"));
    // ... aggiunge il pagamento solo se la fattura appartiene al caller
}
```

InvoiceRepository.java — metodi hotel-scoped

```java
// Lookup singola fattura con check ownership (Spring Data query derivata)
Optional<Invoice> findByIdAndHotelId(UUID id, UUID hotelId);

// Lista paginata filtrata per hotel
Page<Invoice> findByHotelId(UUID hotelId, Pageable pageable);

// Ultima fattura per prenotazione, scoped per hotel
Optional<Invoice> findFirstByReservationIdAndHotelIdOrderByIssueDateDesc(
    UUID reservationId, UUID hotelId);
```

Test Aggiunti/aggiornati test in `InvoiceServiceImplTest` e `PaymentServiceImplTest`: il `SecurityContextHolder` viene inizializzato in `setUp()` con un `UsernamePasswordAuthenticationToken` il cui `details` è il `hotelId` del caller, e ripulito in `tearDown()`. Aggiunto il caso `shouldThrowWhenInvoiceBelongsToDifferentHotel` che verifica il comportamento IDOR-safe (la query hotel-scoped restituisce `Optional.empty()` → HTTP 404).

Riferimento Commit: `fix(security): T-BILL-01 -- IDOR fix on billing-service (hotel-scoped invoice/payment lookup)`

3.7.4 Data Leak Cross-Tenant sul Report Finanziario Owner [T-BILL-04]

Descrizione Trovato durante il design di una regola ArchUnit per imporre lo scoping `hotel_id` su tutte le query multi-tenant (non da un pentest mirato) — l'endpoint `GET /api/v1/reports/owner`, protetto da `@PreAuthorize("hasAnyRole('OWNER','ADMIN')")`, non estraeva mai l'`hotelId` dal contesto di sicurezza. Il controller passava le sole date al service, che interrogava `InvoiceRepository.findByIssueDateBetween(start, end)` — una query priva di qualunque parametro `hotel_id`. Il report aggregava fatturato, conteggio fatture e lista fatture (inclusi dati ospite) di *tutti* gli hotel della piattaforma. Non serviva alcuna enumerazione di UUID: era il comportamento di default dell'endpoint per qualunque Owner/Admin autenticato, su un hotel qualsiasi. Il bug era invisibile in produzione con un solo hotel attivo, ma immediatamente sfruttabile non appena la piattaforma passa multi-tenant. Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — OwnerReportController/ServiceImpl.java (baseline)

```java
// INSICURO: nessun hotelId estratto dal contesto autenticato
@GetMapping("/owner")
@PreAuthorize("hasAnyRole('OWNER', 'ADMIN')")
public ResponseEntity<OwnerFinancialReportDto> getOwnerFinancialReport(
        LocalDate startDate, LocalDate endDate) {
    OwnerFinancialReportDto report =
        ownerReportService.getFinancialReport(startDate, endDate); // no hotel
            ↪ scope
    return ResponseEntity.ok(report);
}

// INSICURO: aggrega fatture di TUTTI gli hotel
public OwnerFinancialReportDto getFinancialReport(LocalDate startDate, LocalDate
    ↪  endDate) {
    List<Invoice> invoices = invoiceRepository.findByIssueDateBetween(start, end)
        ↪ ;
    // ... nessun filtro hotel_id ...
}
```

Mitigazione applicata Stesso pattern già consolidato per T-BILL-01/T-GST-01/T-RES-02, applicato a un endpoint mai coperto in precedenza:

1. **extractHotelId() nel controller** — stesso helper già presente in `InvoiceController`, legge `Authentication.details` (popolato dall'header `X-Auth-Hotel` via `InternalAuthFilter`).
2. **Nuovo metodo repository hotel-scoped** — `findByHotelIdAndIssueDateBetween(hotelId, start, end)` sostituisce l'unscoped `findByIssueDateBetween`.
3. **Firma del service estesa** — `getFinancialReport(hotelId, startDate, endDate)`; l'`hotelId` arriva sempre dal contesto autenticato, mai dal client.
4. **Pulizia correlata** — rimosso anche `InvoiceRepository.findByGuestId` (dead code, zero chiamanti, stesso rischio strutturale se mai richiamato in futuro).

Codice Sicuro — OwnerReportController/ServiceImpl.java (fix)

```java
// SICURO: hotelId risolto dal SecurityContext, mai dal client
@GetMapping("/owner")
@PreAuthorize("hasAnyRole('OWNER', 'ADMIN')")
public ResponseEntity<OwnerFinancialReportDto> getOwnerFinancialReport(
        LocalDate startDate, LocalDate endDate) {
    UUID hotelId = extractHotelId();
    OwnerFinancialReportDto report =
        ownerReportService.getFinancialReport(hotelId, startDate, endDate);
    return ResponseEntity.ok(report);
}

// SICURO: aggrega solo le fatture dell'hotel del caller
public OwnerFinancialReportDto getFinancialReport(
        UUID hotelId, LocalDate startDate, LocalDate endDate) {
    List<Invoice> invoices =
        invoiceRepository.findByHotelIdAndIssueDateBetween(hotelId, start, end);
    // ...
}
```

Test Nuovo `shouldScopeQueryToTheAuthenticatedHotelOnly` in `OwnerReportServiceImplTest` verifica che il repository sia sempre invocato con l'`hotelId` del caller. `OwnerReportControllerTest` aggiornato con `SecurityContextHolder` popolato (stesso pattern di T-BILL-01), asserzioni `verify(...)` sull'`hotelId` propagato.

Riferimento Commit: `fix(security): T-BILL-04 -- cross-tenant data leak on Owner financial report`

3.7.5 Denial of Service su Generazione PDF Fattura [T-BILL-05]

Descrizione `ThymeleafPdfTemplateRenderer` (modulo condiviso `pdf-template-engine`, usato da billing-service per generare il PDF fattura) rileggeva e ri-subsettava i font Noto Sans dal classpath ad *ogni* chiamata a `render()` — l'unico punto di allocazione Metaspace per-richiesta in tutta la pipeline di generazione. Nessun limite era imposto al numero di generazioni concorrenti o sequenziali: un volume anche moderato di richieste (traffico legittimo, non uno scenario di attacco mirato) esauriva progressivamente il Metaspace e crashava la JVM del servizio — CWE-400, uncontrolled resource consumption, classificabile OWASP A04:2021 — Insecure Design (assenza di un limite di capacità progettato, non una misconfigurazione puntuale). Il sintomo è rimasto a lungo mascherato da un secondo bug indipendente e concorrente (`GuestClient`, il client Feign billing→guest-service, privo di configurazione discovery: ogni chiamata falliva con "No servers available" e ogni fattura mostrava "Unknown Guest"), che riduceva drasticamente quanto la funzionalità venisse effettivamente esercitata prima della sua riparazione.

ThymeleafPdfTemplateRenderer.java — font ricaricati ad ogni render() (baseline)

```java
// INSICURO: ogni chiamata a render() rilegge e ri-subsetta i font dal classpath
```

```java
public byte[] render(final String templateName, final Map<String, Object> model)
    ↪ {
    final PdfRendererBuilder builder = new PdfRendererBuilder();
    // font caricati per-richiesta: allocazione Metaspace non delimitata
    builder.useFont(() -> loadFontFromClasspath("NotoSans-Regular.ttf"), "Noto
        ↪ Sans");
    builder.useFont(() -> loadFontFromClasspath("NotoSans-Bold.ttf"), "Noto Sans"
        ↪ , 700);
    // ... render ...
}
```

Mitigazione applicata

1. **Font caricati una sola volta** — letti in campi statici all'inizializzazione della classe, non più ad ogni `render()`.
2. **GuestClient pinnato a URL esplicito** — nuova proprietà `application.config.guest-servic` in config-service, stesso pattern già in uso per `HotelSettingsClient/frontdesk-service-url` (Eureka è disabilitato cluster-wide nel progetto, la discovery automatica non è mai stata disponibile).
3. **Rete di sicurezza operativa** — `MaxMetaspaceSize` alzato da 128m a 192m come margine addizionale, non come mitigazione primaria.

ThymeleafPdfTemplateRenderer.java — font caricati una sola volta (fix)

```java
// SICURO: font letti una sola volta, riusati su ogni render()
private static final FSSupplier<InputStream> NOTO_SANS_REGULAR =
        loadFontFromClasspath("NotoSans-Regular.ttf");
private static final FSSupplier<InputStream> NOTO_SANS_BOLD =
        loadFontFromClasspath("NotoSans-Bold.ttf");

public byte[] render(final String templateName, final Map<String, Object> model)
    ↪ {
    final PdfRendererBuilder builder = new PdfRendererBuilder();
    builder.useFont(NOTO_SANS_REGULAR, "Noto Sans");
    builder.useFont(NOTO_SANS_BOLD, "Noto Sans", 700);
    // ... render ...
}
```

Verifica Dal vivo: rebuild e restart di config-server e billing-service, 20 generazioni PDF consecutive → 20×200, nome ospite reale risolto correttamente, nessun crash, Metaspace stabile sotto il nuovo cap.

Riferimento Commit: `fix(billing): risolve crash OOM su generazione PDF fattura`

3.7.6 Data Leak Cross-Tenant sul Pre-fill Check-in [T-STAY-06]

Descrizione Trovato nella stessa esplorazione di T-BILL-04. `StayServiceImpl.getLastCompletedSt` — usato per pre-compilare il form di check-in con i dati dell'ultimo soggiorno di un ospite di

ritorno — interrogava `StayRepository.findTopByGuestIdAndStatusOrderByActualCheckInTimeDes` `CHECKED_OUT)` senza alcun parametro `hotel_id`. L'unico controllo presente era che il profilo ospite fosse ancora attivo su guest-service, non che il soggiorno restituito appartenesse all'hotel del chiamante. Un receptionist dell'Hotel A, per un `guestId` noto, poteva ottenere numero camera, date e stato fattura/Alloggiati dell'ultimo soggiorno di quell'ospite anche se avvenuto presso l'Hotel B. Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — StayServiceImpl.java (baseline)

```
// INSICURO: nessun hotelId, solo verifica che il guest sia attivo
public Optional<StayResponse> getLastCompletedStayForGuest(UUID guestId) {
    GuestResponse guest = guestClient.getGuestById(guestId); // solo attivo?
    return stayRepository
        .findTopByGuestIdAndStatusOrderByActualCheckInTimeDesc(guestId,
            ↪ CHECKED_OUT)
        .map(stayMapper::toDto); // soggiorno di QUALSIASI hotel
}
```

Mitigazione applicata Stesso pattern di T-STAY-04/T-STAY-05: nuova query hotel-scoped `findTopByGuestIdAndHotelIdAndStatusOrderByActualCheckInTimeDesc`; il service e il controller richiedono ora `hotelId`, risolto da `StayController.extractHotelId()` (mai dal client).

Codice Sicuro — StayServiceImpl.java (fix)

```
// SICURO: hotelId propagato dal controller, mai dal client
public Optional<StayResponse> getLastCompletedStayForGuest(UUID guestId, UUID
    ↪ hotelId) {
    GuestResponse guest = guestClient.getGuestById(guestId);
    return stayRepository
        .findTopByGuestIdAndHotelIdAndStatusOrderByActualCheckInTimeDesc(
            guestId, hotelId, CHECKED_OUT)
        .map(stayMapper::toDto);
}
```

Test Nuovi `shouldReturnLastCompletedStayScopedToTheAuthenticatedHotel` e `shouldReturnEmptyWhenGuestServiceUnavailableForPreFill` in `StayServiceImplTest` — l'endpoint non aveva alcuna copertura di regressione in precedenza, coerente con come il gap sia passato inosservato.

Riferimento Commit: `fix(security): T-STAY-06 -- cross-tenant data leak on check-in pre-fill`

3.7.7 IDOR e Tampering Cross-Tenant sul Catalogo Room Type [T-ROOM-02]

Descrizione La tabella `room_types` risale alla baseline V1, precedente alla convenzione `hotel_id` adottata per l'isolamento multi-tenant, ed era rimasta priva della colonna. Il

gap era stato inizialmente registrato in GAP-6 come limite architetturale accettato ("catalogo tipologie camera globale, non per-hotel"); un approfondimento successivo ha invece dimostrato che si trattava di un IDOR reale e sfruttabile: qualunque ADMIN/OWNER autenticato di un hotel, noto un UUID di room-type di un hotel diverso, poteva leggerlo, modificarne prezzo/nome/capienza o cancellarlo (soft-delete) — non solo un filtro di lettura mancante, ma superficie di scrittura cross-tenant. Il vincolo `UNIQUE(name)` globale impediva inoltre a due hotel indipendenti di usare entrambi lo stesso nome di tipologia (es. "Deluxe"). Stessa classe di T-BILL-04/T-STAY-06, trovato con lo stesso metodo (verifica dei gap noti della regola ArchUnit di GAP-6). Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — RoomTypeServiceImpl.java (baseline)

```java
// INSICURO: nessun hotelId, room_types senza colonna hotel_id
public RoomTypeResponse getRoomTypeById(UUID id) {
    RoomType roomType = roomTypeRepository.findById(id)
        .orElseThrow(() -> new NotFoundException("ROOM_TYPE_NOT_FOUND"));
    return roomTypeMapper.toResponse(roomType); // di QUALSIASI hotel
}

public RoomTypeResponse updateRoomType(UUID id, RoomTypeRequest request) {
    RoomType roomType = roomTypeRepository.findById(id)
        .orElseThrow(() -> new NotFoundException("ROOM_TYPE_NOT_FOUND"));
    // scrive su una room-type di un hotel arbitrario
    roomType.setName(request.name());
    roomType.setBasePrice(request.basePrice());
    return roomTypeMapper.toResponse(roomTypeRepository.saveAndFlush(roomType));
}
```

Mitigazione applicata Flyway V7 aggiunge `hotel_id UUID NOT NULL` a `room_types` (backfill al placeholder esistente, poi `NOT NULL`) e sostituisce `UNIQUE(name)` con `UNIQUE(hotel_id, name)`. `RoomTypeRepository` guadagna `findByIdAndHotelId/findAllByHotelId`; tutti i 5 metodi di `RoomTypeService` richiedono ora `hotelId`, risolto da `RoomTypeController.resolveH` (stesso pattern di `RoomController`), mai dal client. Le chiavi della cache Spring sono state ri-namespaced per hotelId (`#hotelId + ':' + #id` e `#hotelId`) per evitare cache poisoning cross-tenant. Fix connesso individuato nello stesso giro: `RoomServiceImpl` chiamava `roomTypeRepository.findById(roomTypeId)` non scoped in `createRoom/updateRoom` — un consumer del repository, non catturato dalla regola ArchUnit di GAP-6 che analizza solo i metodi dichiarati sul repository stesso, non i call-site dei suoi metodi ereditati. Sostituito con `findByIdAndHotelId`. `RoomTypeRepository` aggiunto ai `TENANT_ROOT_REPOSITORIES` di `TenantIsolationArchTest`.

Codice Sicuro — RoomTypeServiceImpl.java (fix)

```java
// SICURO: hotelId propagato dal controller, mai dal client
public RoomTypeResponse getRoomTypeById(UUID id, UUID hotelId) {
    RoomType roomType = roomTypeRepository.findByIdAndHotelId(id, hotelId)
        .orElseThrow(() -> new NotFoundException("ROOM_TYPE_NOT_FOUND"));
    return roomTypeMapper.toResponse(roomType);
}
```

```
public RoomTypeResponse updateRoomType(UUID id, UUID hotelId, RoomTypeRequest
    ↪ request) {
    RoomType roomType = roomTypeRepository.findByIdAndHotelId(id, hotelId)
        .orElseThrow(() -> new NotFoundException("ROOM_TYPE_NOT_FOUND"));
    roomType.setName(request.name());
    roomType.setBasePrice(request.basePrice());
    return roomTypeMapper.toResponse(roomTypeRepository.saveAndFlush(roomType));
}
```

Test `RoomTypeServiceImplTest` estesa con 2 regressioni cross-hotel (`testGetRoomTypeByIdBelongsT` `testUpdateRoomTypeBelongsToDifferentHotelReturnsNotFound`); `RoomTypeControllerTest` estesa con `SecurityContext` hotel-scoped (pattern `RoomControllerTest`); nuovo `RoomTypeServiceIntegrationTest` contro PostgreSQL reale (Testcontainers) valida la migration V7 end-to-end: due hotel possono usare lo stesso nome di room-type, e nessuno dei tre metodi (get/update/delete) espone o modifica una room-type di un hotel diverso.

Riferimento Commit: `fix(security): T-ROOM-02 -- cross-tenant IDOR and tampering on room types`

3.7.8 Bypass di Autenticazione sul Canale SSE Realtime via Collisione HandlerMapping [T-GW-11]

Descrizione Trovato in security review prima del merge (non da un audit post-hoc): il nuovo endpoint SSE `GET /api/v1/events/stream` (sincronizzazione realtime dello stato camere/check-in/check-out, `api-gateway`, primo uso di SSE nel backend) era raggiunto tramite una route self-referenziale — `uri: forward:/api/v1/events/stream` in `api-gateway.yml` — che mappava `EventStreamController` sulla **stessa** path pubblica su cui la route stessa predica. L'intento era che `forward:` facesse comunque passare la richiesta per `AuthenticationFilter` e i quattro `GlobalFilter` del gateway, esattamente come ogni endpoint proxato.

Non è così in questa applicazione: `DispatcherHandler` di Spring interroga i bean `HandlerMapping` in ordine crescente di `@Order` e delega al primo che produce un match. `RequestMappingHandlerMapping` di WebFlux (che risolve i controller `@GetMapping`, incluso questo) si registra all'ordine 0; `RoutePredicateHandlerMapping` di Spring Cloud Gateway si registra all'ordine 1. Poiché il controller era mappato sulla stessa path su cui la route predicava, ogni richiesta diretta a quella path veniva risolta dal primo mapping — il controller stesso — e la catena di filtri della route non veniva mai raggiunta: `AuthenticationFilter` non veniva mai eseguito. Un chiamante privo di qualunque autenticazione (nessun cookie JWT) poteva inviare `GET /api/v1/events/stream` con un header `X-Auth-Hotel` arbitrario e ricevere lo stream realtime di un hotel qualsiasi. Non un rischio residuo o teorico: bypass completo di autenticazione più IDOR cross-tenant sul canale, verificato riproducendo il percorso di dispatch con `RequestMappingHandlerMapping` in isolamento. Classificato OWASP A01 — Broken Access Control.

Codice Vulnerabile — EventStreamController.java + api-gateway.yml (baseline)

```java
// INSICURO: il controller e la route puntano alla STESSA path pubblica --
// RequestMappingHandlerMapping (order 0) intercetta la richiesta prima che
// RoutePredicateHandlerMapping (order 1) possa applicare AuthenticationFilter.
@GetMapping(path = "/api/v1/events/stream", ...)
public Flux<ServerSentEvent<RoomEvent>> stream(...) { ... }
```

```yaml
- id: events-stream
  uri: forward:/api/v1/events/stream # stessa path del predicate sotto!
  predicates:
    - Path=/api/v1/events/stream
  filters:
    - AuthenticationFilter           # mai eseguito da una richiesta diretta
```

Codice Sicuro — EventStreamController.java + api-gateway.yml (fix)

```java
// SICURO: il controller vive su una path interna, mai esposta. Nessun
// @GetMapping risolve piu' /api/v1/events/stream, quindi solo
// RoutePredicateHandlerMapping (e quindi AuthenticationFilter) puo' risolverla.
public static final String INTERNAL_STREAM_PATH = "/local/events/stream";

@GetMapping(path = INTERNAL_STREAM_PATH,
        produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<ServerSentEvent<RoomEvent>> stream(
        @RequestHeader(value = "X-Auth-Hotel", required = false)
        String hotelHeader) {
    UUID hotelId = parseHotelId(hotelHeader); // 401 se assente/malformato
    return broadcaster.streamFor(hotelId) // un solo Sinks.Many per hotelId
            .map(event -> ServerSentEvent.builder(event).build());
}
```

```yaml
- id: events-stream
  uri: forward:/local/events/stream # path DIVERSA dal predicate pubblico
  predicates:
    - Path=/api/v1/events/stream
  filters:
    - AuthenticationFilter           # ora l'unico modo di raggiungere il
      ↪ controller
```

Mitigazione applicata

1. Separazione strutturale dei due path: la path pubblica (`/api/v1/events/stream`, su cui la route predica) e la path interna a cui il controller è realmente mappato (`/local/events/stream`) non coincidono mai. Poiché nessun `@GetMapping` risolve più la path pubblica, `RequestMappingHandlerMapping` non può intercettarla — resta l'unica strada `RoutePredicateHandlerMapping`, che applica sempre `AuthenticationFilter` prima di inoltrare.

2. Nuovo `EventStreamControllerMappingTest`: asserisce via reflection che la path del `@GetMapping` non sia mai uguale alla path pubblica hardcoded nel test stesso (de-

liberatamente non derivata dalla stessa costante che una regressione modificherebbe) — guardia di regressione per esattamente questa classe di bug.

3. L'hotelId resta comunque letto esclusivamente dall'header `X-Auth-Hotel`, iniettato da `AuthenticationFilter` dal claim JWT già verificato, mai da un parametro di query o da un header non firmato che il client potrebbe manipolare direttamente.

4. `RoomEventBroadcaster` mantiene un `ConcurrentHashMap<UUID, Sinks.Many<RoomEvent>>`, un sink per hotelId: anche assumendo un chiamante autenticato, un tenant può ottenere soltanto il `Flux` del proprio hotel.

5. Il payload dell'evento (`RoomEvent`) è deliberatamente minimale — solo tipo e timestamp — il frontend invalida la cache React Query e rifà il fetch autenticato, invece di fidarsi di dati applicativi ricevuti sul canale push.

Test `EventStreamControllerMappingTest` (nuovo, la regressione specifica di questo finding): la path del controller non coincide mai con quella pubblica. `EventStreamControllerTest`: 401 con header hotel assente o non-UUID; un evento pubblicato per un hotelId arriva solo agli iscritti di quello stesso hotelId, non a quelli di un hotelId diverso (test dedicato con due sink paralleli). `RoomEventBroadcasterTest` verifica lo stesso isolamento a livello di sink, indipendentemente dal layer HTTP.

Riferimento Commit: `feat(secure-coding-hardening): SSE realtime room-status sync (T-GW-11, T-GW-12)`

3.7.9 Verifica HMAC in Ingresso sul Gateway [T-GW-12]

Descrizione Anche T-GW-12 è una minaccia modellata proattivamente, non una vulnerabilità trovata in codice esistente. Il nuovo endpoint `POST /internal/events/notify` (`frontdesk-service` → `api-gateway`, chiamata interna che innesca la pubblicazione di un evento sul canale SSE di T-GW-11) è il primo caso in cui il gateway *riceve*, e non solo produce, una chiamata interna firmata HMAC. Il verificatore condiviso da tutti gli altri sei servizi, `InternalAuthFilter` (`internal-auth-lib`, vedi T-GW-08), estende `OncePerRequestFilter` ed è quindi legato allo stack servlet — non riusabile as-is su `api-gateway`, che è reattivo (WebFlux/Netty). Il rischio: un verificatore reimplementato in modo indipendente potrebbe divergere dall'algoritmo condiviso (finestra di tolleranza sul timestamp, confronto non a tempo costante, nessun controllo anti-replay sul nonce), indebolendo silenziosamente la garanzia che solo il gateway stesso possa autenticare chiamate interne (T-GW-01/T-GW-08).

Codice Sicuro — InternalEventsAuthFilter.java

```java
// SICURO: stesso algoritmo byte-per-byte di InternalAuthFilter, reimplementato
// per lo stack reattivo -- stesso payload, stessa finestra di tolleranza,
// stesso confronto a tempo costante, stesso meccanismo anti-replay.
private boolean isSignatureValid(String username, String role, String hotelId,
        String timestamp, String nonce, String signature) {
    String expected = computeHmac(username, role, hotelId, timestamp, nonce);
    return MessageDigest.isEqual(
            expected.getBytes(StandardCharsets.UTF_8),
            signature.getBytes(StandardCharsets.UTF_8));
```

```
}

// Nonce store reattivo: il gateway ha solo lo stack Redis reattivo,
// non quello bloccante da cui dipende RedisNonceStore.
private Mono<Boolean> claimNonce(String nonce) {
    return redisTemplate.opsForValue().setIfAbsent(
            "internal-auth:nonce:" + nonce, "1",
            Duration.ofSeconds(NONCE_TTL_SECONDS));
}
```

Mitigazione applicata

1. Payload HMAC identico a `InternalAuthFilter/AuthenticationFilter.computeHmac`: `HMAC-SHA256(secret, "user:role:hotelId:timestamp:nonce")`, esadecimale minuscolo.
2. Finestra di tolleranza identica: ±60 secondi (T-GW-08).
3. Nonce store reattivo (`ReactiveStringRedisTemplate.setIfAbsent`, semantica SETNX) con TTL 120s, stessa politica di `RedisNonceStore` ma senza condividerne il codice — prima parte reattiva del backend, forzare un'astrazione comune servlet/reactive sarebbe stata over-engineering prematuro.
4. Lato `frontdesk-service` (chiamante), zero codice nuovo di firma: `InternalFeignAuthIntercept` è già registrato come bean globale in `FeignHeaderConfig` e firma automaticamente ogni nuovo client Feign, incluso `GatewayEventsClient`.

Test `InternalEventsAuthFilterTest` replica l'intera matrice di `InternalAuthFilterTest`: header mancanti, firma non valida, firma calcolata per un hotelId diverso (isolamento tenant), timestamp scaduto, replay dello stesso nonce rifiutato.

Riferimento Commit: `feat(secure-coding-hardening): SSE realtime room-status sync (T-GW-11, T-GW-12)`

3.8 Gestione Segreti

3.8.1 Rimozione segreti in chiaro [T-CFG-02]

Descrizione Nella baseline il sistema presentava due categorie di segreti in chiaro nei file versionati:

1. **JWT_SECRET con fallback hardcoded** in `config-service/src/main/resources/config/auth-service.yml`: il valore di default `bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==` (base64 della stringa nota `my-32-byte-secret-key-for-test-local-dev-12345`) era committato nel repository. Se la variabile d'ambiente `JWT_SECRET` non veniva impostata, il servizio avviava silenziosamente con questo segreto debole e noto. Un attaccante con accesso al repository — o che ne conosce la struttura — poteva forgiare JWT validi per qualsiasi utente.

2. **POSTGRES_PASSWORD hardcoded** a `password` nel `docker-compose.yml` e in tutti e 7 i blocchi `SPRING_DATASOURCE_PASSWORD`: la password del database era in chiaro in un file versionato, identica per tutti gli ambienti, senza possibilità di rotazione senza modifica del codice sorgente.

Classificato OWASP A04:2025 — Cryptographic Failures.

Configurazione Vulnerabile — auth-service.yml (baseline)

```
# INSICURO: fallback hardcoded --se JWT_SECRET non e' impostata,
# viene usato un segreto debole e noto committato nel repository
jwt:
  secret: "${JWT_SECRET:
      ↪ bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==}"
```

docker-compose.yml (baseline) — segreti DB in chiaro

```
postgres:
  environment:
    POSTGRES_PASSWORD: password # hardcoded, identico in tutti gli ambienti

auth-service:
  environment:
    - SPRING_DATASOURCE_PASSWORD=password # idem su tutti e 7 i servizi
```

Mitigazione applicata L'intervento si articola in tre modifiche coordinate:

1. **Rimozione del fallback** da `auth-service.yml`: il valore di default è stato eliminato. La proprietà diventa `"${JWT_SECRET}"` senza alternativa: se la variabile d'ambiente non è presente, Spring Boot fallisce all'avvio con un errore esplicito (*fail-fast*), impedendo che il servizio parta con un segreto insicuro.

2. **Parametrizzazione di `POSTGRES_PASSWORD`** nel `docker-compose.yml`: la password del database è ora `"${POSTGRES_PASSWORD}"` sia per il container PostgreSQL sia per tutti e 7 i servizi applicativi. Il valore effettivo viene caricato esclusivamente dal file `.env` (gitignored).

3. **Aggiornamento degli script di setup** (`setuphmacsecret.sh` e `setuphmacsecret.ps1`): entrambi gli script generano ora, in modo idempotente, anche `JWT_SECRET` (48 byte random in Base64) e `POSTGRES_PASSWORD` (16 byte random in esadecimale) nel file `.env`, oltre al già gestito `INTERNAL_HMAC_SECRET`.

Configurazione Sicura — auth-service.yml (feature/secure-coding-hardening)

```
# SICURO: nessun fallback --il servizio fallisce all'avvio
# se JWT_SECRET non e' impostata nell'ambiente (fail-fast)
jwt:
  secret: "${JWT_SECRET}"
```

docker-compose.yml (post-hardening) — segreti da variabili d'ambiente

```yaml
postgres:
  environment:
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD} # da .env (gitignored)

auth-service:
  environment:
    - SPRING_DATASOURCE_PASSWORD=${POSTGRES_PASSWORD}
    - JWT_SECRET=${JWT_SECRET}            # iniettato nel container, non nel repo
```

setup-hmac-secret.sh (estratto) — generazione idempotente di tutti i segreti

```bash
# Genera JWT_SECRET se non gia' presente in .env
if ! grep -q "^JWT_SECRET=" "$ENV_FILE"; then
    JWT_SECRET="$(openssl rand -base64 48 | tr -d '\n')"
    printf "JWT_SECRET=%s\n" "$JWT_SECRET" >> "$ENV_FILE"
fi

# Genera POSTGRES_PASSWORD se non gia' presente in .env
if ! grep -q "^POSTGRES_PASSWORD=" "$ENV_FILE"; then
    POSTGRES_PASSWORD="$(openssl rand -hex 16)"
    printf "POSTGRES_PASSWORD=%s\n" "$POSTGRES_PASSWORD" >> "$ENV_FILE"
fi
```

Il file `.env` rimane gitignored e viene generato localmente da ogni sviluppatore tramite lo script di setup, oppure iniettato dal sistema di CI/CD come secret di ambiente. Nessun segreto transita più attraverso il repository.

Riferimento Commit: `fix(security): T-CFG-02 -- remove JWT_SECRET fallback, parameterize POSTGRES_PASSWORD in docker-compose`

Gap residuo — fallback in JwtUtil dell'API Gateway L'intervento descritto sopra rimuoveva il fallback soltanto da `auth-service.yml`. Un secondo segreto debole era rimasto nascosto in `api-gateway/src/main/java/com/hotelpms/gateway/util/JwtUtil.java`: l'annotazione `@Value` conteneva lo stesso default Base64 hardcoded, rendendo di fatto inefficace la protezione se l'API Gateway veniva avviato senza la variabile d'ambiente.

JwtUtil.java (api-gateway) — fallback hardcoded residuo

```java
// INSICURO: il gateway accetta JWT firmati con il segreto debole
// se JWT_SECRET non e' presente nell'ambiente
@Value("${jwt.secret:
    ↪ bXktMzItYnl0ZS1zZWNyZXQta2V5LWZvci10ZXN0LWxvY2FsLWRldi0xMjM0NQ==}")
private String secret;
```

api-gateway.yml (baseline) — proprieta' jwt.secret assente

```
# jwt.secret non configurato: il gateway legge il fallback dalla
# annotazione @Value nel codice sorgente (segreto debole e noto)
internal:
  hmac:
    secret: ${INTERNAL_HMAC_SECRET}
```

Seconda mitigazione applicata

1. **Rimozione del default dall'annotazione `@Value`**: il parametro `jwt.secret` non ha più valore alternativo. L'assenza della variabile d'ambiente causa un fallimento esplicito all'avvio del gateway (*fail-fast*), allineando il comportamento a quello già in vigore per `auth-service`.

2. **Aggiunta di `jwt.secret` in `api-gateway.yml`**: la chiave viene ora dichiarata esplicitamente nel file di configurazione centralizzata, con il binding obbligatorio `${JWT_SECRET}`. In questo modo il valore viene risolto dall'ambiente (Docker secret o file `.env`) in modo coerente con tutti gli altri segreti del sistema.

JwtUtil.java (post-hardening) — fail-fast senza fallback

```java
// SICURO: nessun default --il container non si avvia se JWT_SECRET
// non e' definita; nessun segreto debole puo' essere usato silenziosamente
@Value("${jwt.secret}")
private String secret;
```

api-gateway.yml (post-hardening) — jwt.secret esplicito da env

```
jwt:
  secret: "${JWT_SECRET}" # obbligatorio: fail-fast se assente

internal:
  hmac:
    secret: ${INTERNAL_HMAC_SECRET}
```

Riferimento Commit: `fix(security): T-CFG-02 -- rimuovi fallback JWT_SECRET hardcoded da api-gateway JwtUtil`

3.8.2 Regressione della guardia legale GDPR TULPS/fiscale [T-GST-06]

Descrizione Trovato dal secondo giro di test esplorativo dal vivo (`docs/EXPLORATORY_TEST_2026-08.` Round 2 bug #1). La guardia legale già corretta con T-GST-05 — che deve bloccare l'anonimizzazione GDPR di un ospite finché sono attivi vincoli TULPS (5 anni) o fiscali (10 anni) — si era di nuovo rotta, con una tripla causa indipendente accumulata su refactoring successivi (consolidamento ADR-001, introduzione di `@EnableMethodSecurity`). Riprodotto dal vivo su un ospite reale con due fatture `PAID` del 2026, ben dentro la

finestra fiscale decennale: `DELETE /guests/{id}` restituiva `204 No Content` invece del `451` atteso, distruggendo in modo irreversibile l'identità dell'ospite nonostante l'obbligo di conservazione fosse ancora attivo. Classificato OWASP A04:2025 — Insecure Design: tre difetti indipendenti che si compensavano a vicenda nel non-far-scattare mai il blocco, nessuno dei quali un singolo bug ovvio isolatamente.

1. **Path Feign errato**: `BillingServiceClient.getLastInvoiceDate` chiamava `/api/v1/invoices/guest/{guestId}/last-invoice-date`, ma la rotta reale del controller è `.../last-date` — 404 garantito ad ogni chiamata.

2. **Metodi `final` sotto proxy CGLIB**: `InvoiceServiceImpl.getLastInvoiceDateForGuest` e `StayServiceImpl.getLastStayDateForGuest` erano gli unici due metodi `public final` nelle rispettive classi. Con `proxy-target-class=true` (introdotto da `@EnableMethodSecurity`, T-FB-03/GAP-8), Spring genera un proxy CGLIB per sottoclassamento: un metodo `final` non è overridabile dalla sottoclasse proxy, quindi la chiamata esegue il bytecode originale su un'istanza-guscio mai inizializzata via dependency injection — `NullPointerException` garantita su `stayRepository`/`invoiceRepository`.

3. **Fallback dichiarato fail-safe, di fatto fail-open**: il fallback del circuit breaker (Javadoc: "fail-safe... block deletion") ritorna `hasStays/hasInvoices=true` con data `null`. Ma la logica che lo consuma (`computeTulpsExpiry`/`computeFiscalExpiry` in `GuestServiceImpl`, `shouldAnonymise` in `GuestRetentionJobServiceImpl`) leggeva `lastStayDate()==null` come "nessun vincolo" — esattamente l'esito opposto a quello dichiarato nel commento.

Codice Vulnerabile — GuestServiceImpl.java (prima del fix)

```java
// INSICURO: se il fallback ritorna hasStays=true con data null
// (dependent service degradato), tulpsExpiry resta null e il vincolo
// e' letto come "nessun blocco" -- l'opposto del fail-safe dichiarato
private static LocalDate computeTulpsExpiry(
        GuestLastStayClientResponse info, int retentionYears) {
    if (!info.hasStays() || info.lastStayDate() == null) {
        return null;  // <-- qui: indeterminato trattato come "libero"
    }
    return info.lastStayDate().plusYears(retentionYears);
}
```

Mitigazione applicata Tre correzioni indipendenti, tutte necessarie contemporaneamente: (1) path Feign allineato a `/last-date`; (2) `final` rimosso da entrambi i metodi (con `/** {@inheritDoc} */` per soddisfare la regola Checkstyle `DesignForExtension` sul metodo ora overridabile); (3) `verifyLegalHolds` e `shouldAnonymise` ora distinguono esplicitamente il caso indeterminato (`hasStays()=true` con data `null`) dal caso "nessun vincolo" (`hasStays()=false`), trattando il primo come bloccante.

Codice Sicuro — GuestServiceImpl.java (fix)

```java
// SICURO: l'indeterminazione blocca, non libera
final boolean tulpsIndeterminate =
```

```
        stayInfo.hasStays() && stayInfo.lastStayDate() == null;
final boolean tulpsBlocked = tulpsIndeterminate
        || tulpsExpiry != null && !LocalDate.now().isAfter(tulpsExpiry);
// analogo per fiscalIndeterminate/fiscalBlocked; l'eccezione 451
// riporta un messaggio esplicito "unable to verify... blocked as
// a precaution" e unlocksAt=null quando la causa e' indeterminazione
```

Test 4 nuovi test boundary-value: due sul guard sincrono (`GuestServiceImplDeleteGuardTest.shou...FiscalStatusIndeterminate`, riproducono esattamente la risposta del fallback) e due sul job notturno (`GuestRetentionJobServiceImplTest.shouldSkipGuestWhenTulpsStatusIndeterm...FiscalStatusIndeterminate`, verificano `guestRepository.save` mai invocato). Verificato dal vivo contro lo stack Docker reale: guest con 2 fatture `PAID` 2026 → `DELETE` → `451 LEGAL_HOLD_ACTIVE` con `unlocksAt=2036-08-04`, `legalBasis=FISCAL`; guest senza storico → `DELETE` → `204` invariato (nessuna regressione sul percorso felice).

Riferimento Commit: `fix(security): T-GST-06 -- GDPR legal-hold guard fail-open regression (Feign path, final under CGLIB, fallback semantics)`

3.9 Analisi SAST con CodeQL e Supply Chain Security [Appendice A]

title=Nota — A03:2025 Software Supply Chain Failures + OWASP SAMM Verification (AI-assisted)

Questa sezione copre **A03:2025 — Software Supply Chain Failures**, categoria nuova dell'edizione 2025 (nel 2021 era solo "Vulnerable and Outdated Components"): l'analisi SAST con GitHub CodeQL, la gestione delle CVE con Trivy e l'hardening CI/CD rientrano esattamente in questa categoria. Corrisponde alla fase **Verification** di OWASP SAMM. Implementata con supporto AI; la critica metodologica è nell'Appendice B.

3.10 GAP-2 — Test E2E Attack-Path (Zero Coverage di Sicurezza)

Descrizione I test E2E **Playwright** esistenti coprono esclusivamente gli *happy path* funzionali (login con successo, creazione prenotazione, check-in). Non esiste alcun test che verifichi il comportamento del sistema quando riceve risposte *di sicurezza* dal backend: 401 Unauthorized, 403 Forbidden, 404 Not Found per IDOR, 429 Too Many Requests per brute-force. Questa assenza lascia scoperti i meccanismi di hardening implementati nelle sezioni precedenti: un bug di regressione nel frontend (ad esempio, un modal che si chiude silenziosamente su 403) non sarebbe rilevato da alcuna suite automatizzata.

Categoria OWASP

- **A01:2025** — Broken Access Control (IDOR, RBAC, accesso non autenticato)

- **A07:2025** — Authentication Failures (JWT scaduto, brute-force 429)
- **A01:2025** — CSRF (403 su mutation)

Baseline — Suite E2E con zero copertura di percorsi di attacco

```
// e2e/auth.spec.ts (baseline)
// Solo happy path: login OK, redirect a dashboard, form visibile.
// Nessun test per:
//   - JWT assente/scaduto (401) -> redirect a /login
//   - Refresh fallito -> performLogout()
//   - 429 Too Many Requests -> errore generico (no detail account)
//   - 403 CSRF -> toast error, modal aperto (no silent success)

// e2e/booking-checkin.spec.ts (baseline)
// Solo happy path: prenotazione -> check-in.
// Nessun test per:
//   - IDOR cross-hotel: PUT /guests/{id} -> 404
//   - URL manipulation: /reservations/edit/{cross-hotel-uuid} -> 404
//   - RBAC: RECEPTIONIST -> /owner-dashboard -> redirect a /
```

Codice Sicuro — 11 test E2E attack-path (3 suite)

```
// -- security-auth.spec.ts ------------------------------------------------

// T-AUTH-SEC-02: sessione scaduta durante la navigazione
test('expired session (401 on data + failed refresh) -> /login', async ({page})
    ↪ => {
  // /me: call #1 -> 200, call #2+ -> 401
  await page.route('**/api/v1/auth/me', async (route) => { /* counter */ });
  // /refresh -> 401 (refresh token revocato o scaduto)
  await page.route('**/api/v1/auth/refresh', route =>
    route.fulfill({ status: 401, body: '{"error":"Unauthorized"}' }));
  // GET /guests -> 401: scatena l'interceptor -> refresh -> logout
  await page.route('**/api/v1/guests', route =>
    route.fulfill({ status: 401, body: '{"error":"Unauthorized"}' }));
  await page.goto('/guests');
  // performLogout() -> window.location.href = '/login'
  await expect(page).toHaveURL(/\/login/, { timeout: 10_000 });
});

// T-AUTH-SEC-03: brute-force rate limit (429) -> errore generico, no detail
test('429 -> generic error, no account detail exposed', async ({ page }) => {
  await page.route('**/api/v1/auth/login', route =>
    route.fulfill({ status: 429, body: '{"detail":"Too Many Requests"}' }));
  await page.goto('/login');
  await page.getByRole('textbox', { name: /username/i }).fill('admin');
  await page.getByLabel(/password/i).fill('wrongpassword');
  await page.locator('[data-testid="login-submit"]').click();
  const errorParagraph =
    page.locator('[data-testid="login-form"] .bg-error-container p');
  await expect(errorParagraph).toBeVisible({ timeout: 5_000 });
  const msg = await errorParagraph.textContent();
  // Nessun dettaglio su lock, ban, tentativi rimasti
  expect(msg).not.toMatch(/lock|ban|remain|attempt/i);
  // Submit ri-abilitato: no UI-level lockout permanente
```

```
  await expect(page.locator('[data-testid="login-submit"]')).toBeEnabled();
});

// T-AUTH-SEC-04: CSRF rejection (403) -> toast error, no silent success
test('403 CSRF rejection -> toast error, modal stays open', async ({ page }) =>
    ↪ {
  await page.route('**/api/v1/guests', async (route) => {
    if (route.request().method() === 'GET')
      await route.fulfill({ status: 200, body: '{"content":[],...}' });
    else // POST senza X-CSRF-Token valido
      await route.fulfill({ status: 403, body: '{"detail":"Forbidden"}' });
  });
  await page.goto('/guests');
  await page.getByRole('button', { name: /add guest/i }).click();
  await expect(page.locator('[role="dialog"]')).toBeVisible();
  await page.locator('input[name="firstName"]').fill('Test');
  await page.locator('input[name="lastName"]').fill('Guest');
  await page.locator('input[name="email"]').fill('test@example.com');
  await page.locator('button[form="guest-form"]').click();
  // Toast error (role=alert da ToastContainer)
  await expect(page.getByRole('alert')).toBeVisible({ timeout: 5_000 });
  // Modal rimane aperto: nessun successo silenzioso
  await expect(page.locator('[role="dialog"]')).toBeVisible();
});

// -- security-rbac.spec.ts ------------------------------------------------

// T-RBAC-SEC-01: RECEPTIONIST -> /owner-dashboard -> ProtectedRoute -> /
test('RECEPTIONIST -> /owner-dashboard -> redirected to /', async ({ page }) =>
    ↪ {
  await page.route('**/api/v1/auth/me', route =>
    route.fulfill({ status: 200,
      body: JSON.stringify({ username: 'r', role: 'RECEPTIONIST' }) }));
  await page.goto('/owner-dashboard');
  await expect(page).toHaveURL(/\/$/, { timeout: 10_000 });
  await expect(
    page.getByRole('heading', { name: /owner dashboard/i })
  ).not.toBeVisible();
});

// -- security-idor.spec.ts ------------------------------------------------

// T-IDOR-SEC-02: URL manipulation su prenotazione cross-hotel -> 404 -> error
test('cross-hotel reservation edit via URL -> error state', async ({ page }) =>
    ↪ {
  const id = 'res-hotel-b-99999';
  await page.route(`**/api/v1/reservations/${id}`, route =>
    route.fulfill({ status: 404,
      body: '{"detail":"RESERVATION_NOT_FOUND"}' }));
  await page.goto(`/reservations/edit/${id}`);
  // loadInitialData() cattura il 404 e chiama setError()
  await expect(
    page.locator('.bg-error-container').first()
  ).toBeVisible({ timeout: 10_000 });
  // Nessun dato di prenotazione popolato
  const dateInputs = page.locator('input[type="date"]');
```

```
  if (await dateInputs.count() > 0)
    await expect(dateInputs.first()).toHaveValue('');
});
```

Copertura ottenuta Le tre suite aggiungono **11 test E2E** che coprono i seguenti vettori:

Suite	Test ID	Vettore di attacco	Assert chiave
security-auth	T-AUTH-SEC-01	Accesso non autenticato	URL = `/login`
security-auth	T-AUTH-SEC-02	Sessione scaduta + refresh fail	performLogout → `/login`
security-auth	T-AUTH-SEC-03	Brute-force (429)	Errore generico, no detail
security-auth	T-AUTH-SEC-04	CSRF (403 su POST)	Toast error, modal aperto
security-rbac	T-RBAC-SEC-01	RECEPTIONIST → route ADMIN	Redirect a `/`
security-rbac	T-RBAC-SEC-02	ADMIN → route ADMIN/OWNER	Accesso concesso
security-rbac	T-RBAC-SEC-03	OWNER → route ADMIN/OWNER	Accesso concesso
security-rbac	T-RBAC-SEC-04	Non autenticato → route prot.	Redirect a `/login`
security-idor	T-IDOR-SEC-01	Edit ospite cross-hotel (404)	Toast error, modal aperto
security-idor	T-IDOR-SEC-02	URL manipulation risorsa (404)	Error state, no dati
security-idor	T-IDOR-SEC-03	Lista ospiti hotel-scoped	Solo ospiti hotel corrente

Riferimento Commit:
`fix: GAP-2 - 11 E2E security attack-path tests (auth/RBAC/IDOR)`

3.11 GAP-3 — fb-service InternalAuthFilter Struttura Inconsistente

Descrizione Il filtro `InternalAuthFilter` del **fb-service** presentava una struttura del *presence check* divergente rispetto agli altri cinque microservizi (guest, billing, reservation, stay, inventory). Mentre il pattern canonico adottato da tutti gli altri servizi verifica la presenza dei quattro header obbligatori in un **singolo** `if` combinato, il fb-service utilizzava **due** blocchi `if` separati: il primo controllava `X-Auth-User`, `X-Auth-Role` e `X-Internal-Signature` (escludendo `X-Auth-Hotel`); il secondo controllava `X-Auth-Hotel` in modo isolato, con un `LOG.warn` dedicato che rivelava il `username` già estratto prima della verifica d'integrità HMAC.
L'impatto diretto sulla sicurezza è contenuto — la firma HMAC include `hotelId` nel payload firmato (`username:role:hotelId`), quindi un header mancante avrebbe comunque causato il fallimento del controllo HMAC. Tuttavia la struttura inconsistente:

- Aumentava la *superficie di manutenzione*: una futura modifica al pattern di un solo servizio sarebbe passata inosservata.

- Violava il principio di **uniformità difensiva**: filtri con struttura diversa rendono il code review meno affidabile (difficile identificare deviazioni dal pattern atteso).
- Il `LOG.warn` nel secondo `if` emetteva il `username` in chiaro anche prima della validazione della firma, benché in un contesto interno non pubblico.

Classificazione OWASP A01 — Broken Access Control (pattern di controllo accessi non uniforme tra i servizi interni).

Codice vulnerabile — fb-service InternalAuthFilter (presenza separata)

```
// fb-service (PRIMA): due if separati -- hotelId escluso dal primo check
if (!StringUtils.hasText(username) || !StringUtils.hasText(role)
        || !StringUtils.hasText(signature)) {      // hotelId assente!
    rejectRequest(response, "Missing required gateway authentication headers");
    return;
}

if (!StringUtils.hasText(hotelId)) {
    // LOG emette username PRIMA della verifica HMAC
    LOG.warn("[InternalAuthFilter] Missing X-Auth-Hotel header for user={}",
        ↪ username);
    rejectRequest(response, "Missing X-Auth-Hotel header");
    return;
}
```

Codice sicuro — fb-service InternalAuthFilter (singolo if canonico)

```
// fb-service (DOPO): singolo if combinato -- identico agli altri 5 servizi
if (!StringUtils.hasText(username) || !StringUtils.hasText(role)
        || !StringUtils.hasText(hotelId) || !StringUtils.hasText(signature)) {
    rejectRequest(response, "Missing required gateway authentication headers");
    return;
}

// LOG.warn rimane solo per mismatch HMAC (dopo che tutti gli header sono
    ↪ presenti)
if (!isSignatureValid(username, role, hotelId, signature)) {
    LOG.warn("[InternalAuthFilter] HMAC signature mismatch for user={}", username
        ↪ );
    rejectRequest(response, "Invalid internal request signature");
    return;
}
```

Test di copertura

Sono stati aggiunti 8 test JUnit 5 in `fb-service/.../security/InternalAuthFilterTest.java` che coprono tutti i path del filtro:

Nested class	Test	Assert chiave
PresenceCheck	shouldRejectWhenUsernameMissing	HTTP 401
PresenceCheck	shouldRejectWhenRoleMissing	HTTP 401
PresenceCheck	shouldRejectWhenHotelIdMissing	HTTP 401
PresenceCheck	shouldRejectWhenSignatureMissing	HTTP 401
HmacCheck	shouldRejectWhenSignatureInvalid	HTTP 401
HmacCheck	shouldRejectWhenSignatureComputedForDifferentHotelId	HTTP 401 (tenant isolation)
HmacCheck	shouldAuthenticateWithValidHeaders	HTTP 200 + SecurityContext (name/details/authority)
ActuatorBypass	shouldBypassFilterForActuatorEndpoint	HTTP 200 senza headers

Riferimento Commit: `fix(security): GAP-3 -- fb-service InternalAuthFilter single combined presence check`

3.12 GAP-4 — Audit Log Non Aggregati (SIEM Assente)

Descrizione I log strutturati emessi dai microservizi con prefissi `[AUTH]`, `[STAY]` e `[BILLING]` venivano scritti esclusivamente su `stdout` del container Docker, senza alcun meccanismo di aggregazione, ricerca o alerting centralizzato. In assenza di un sistema di raccolta, risultava impossibile rispondere a domande operative di sicurezza quali:

- *«Quanti `LOGIN_FAILED` ha prodotto l'utente X nelle ultime 24 ore?»*
- *«Ci sono stati picchi anomali di `ACCOUNT_LOCKED` in una finestra temporale ristretta?»*
- *«Quali richieste hanno fallito la verifica HMAC nelle ultime ore?»*

La mancanza di aggregazione log costituisce una violazione diretta di **OWASP A09 — Security Logging and Monitoring Failures**: anche se i singoli eventi vengono emessi correttamente, la loro utilità è nulla se non possono essere consultati, correlati e alertati in modo centralizzato.

Classificazione OWASP A09 — Security Logging and Monitoring Failures (log prodotti ma non aggregati, nessuna capacità di rilevamento anomalie o risposta a incidenti).

Soluzione implementata La soluzione è composta da tre livelli cooperanti:

1. **Infrastruttura**: aggiunta di **Loki** (`grafana/loki:2.9.0`) come aggregatore di log e **Grafana** (`grafana/grafana:10.2.0`) come layer di visualizzazione e alerting, entrambi nella rete Docker isolata `observability-network` con volumi persistenti.

2. **Appender Logback**: integrazione della libreria `loki-logback-appender:1.5.2` nei tre servizi con audit log strutturati (auth-service, stay-service, billing-service). Ogni servizio riceve un file `logback-spring.xml` dedicato.

3. **Dashboard Grafana**: provisioning automatico di una dashboard *Security Audit Log* con sei pannelli operativi: contatori `LOGIN_FAILED`, `ACCOUNT_LOCKED` e `HMAC Failures` nell'ultima ora, grafico di tendenza degli eventi di sicurezza al minuto, e visualizzazione raw dei log per ciascun prefisso di audit.

Situazione precedente — log su stdout, nessuna aggregazione

Listing 3.1: auth-service: audit event emesso su stdout senza destinazione SIEM

```
// AuthServiceImpl.java (PRIMA) -- @Slf4j, log solo su stdout container
LOG.warn("[AUTH] LOGIN_FAILED user={} reason=BAD_PASSWORD attempt={}/{}",
        username, failedAttempts, lockoutThreshold);
// Il log scompare nel buffer del container; nessun sistema lo raccoglie.
// Impossibile aggregare, cercare, correlare o alertare su questi eventi.
```

Livello 1 — Loki e Grafana in docker-compose (observability-network)

Listing 3.2: docker-compose.yml: servizi Loki e Grafana aggiunti

```
loki:
  image: grafana/loki:2.9.0
  command: -config.file=/etc/loki/config.yaml
  volumes:
    - ./docker/loki/loki-config.yaml:/etc/loki/config.yaml
    - loki_data:/loki
  networks: [observability-network]

grafana:
  image: grafana/grafana:10.2.0
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=admin
    - GF_USERS_ALLOW_SIGN_UP=false
  volumes:
    - ./docker/grafana/provisioning:/etc/grafana/provisioning
    - grafana_data:/var/lib/grafana
  depends_on:
    loki: { condition: service_healthy }
  networks: [observability-network]
```

Livello 2 — logback-spring.xml con LokiAppender (auth-service)

Listing 3.3: logback-spring.xml: Loki attivo solo nel profilo Docker; test usano solo console

```
<configuration>
```

```
  <include resource=
    "org/springframework/boot/logging/logback/defaults.xml"/>
  <springProperty scope="context" name="appName"
    source="spring.application.name" defaultValue="auth-service"/>

  <appender name="CONSOLE"
           class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>${CONSOLE_LOG_PATTERN}</pattern>
    </encoder>
  </appender>

  <!-- Attivo solo con SPRING_PROFILES_ACTIVE=auth-service (Docker) -->
  <springProfile name="auth-service">
    <appender name="LOKI"
             class="com.github.loki4j.logback.Loki4jAppender">
      <http>
        <url>${LOKI_URL:-http://loki:3100}/loki/api/v1/push</url>
      </http>
      <format>
        <label>
          <pattern>service=${appName},level=%level</pattern>
        </label>
        <message>
          <pattern>[%level] [%thread] %logger{36} | %msg%n%ex</pattern>
        </message>
        <sortByTime>true</sortByTime>
      </format>
    </appender>

    <appender name="LOKI_ASYNC"
             class="ch.qos.logback.classic.AsyncAppender">
      <appender-ref ref="LOKI"/>
      <queueSize>512</queueSize>
      <discardingThreshold>0</discardingThreshold>
      <!-- neverBlock: Loki irraggiungibile non blocca mai i thread app -->
      <neverBlock>true</neverBlock>
    </appender>

    <root level="INFO">
      <appender-ref ref="CONSOLE"/>
      <appender-ref ref="LOKI_ASYNC"/>
    </root>
  </springProfile>

  <!-- Profilo default (test / sviluppo locale): solo console -->
  <springProfile name="!auth-service">
    <root level="INFO">
      <appender-ref ref="CONSOLE"/>
    </root>
  </springProfile>
</configuration>
```

Livello 3 — Query LogQL nella dashboard Grafana

Listing 3.4: LogQL: pannelli dashboard "Security Audit Log"

```
# LOGIN_FAILED nell'ultima ora (pannello stat con soglia rossa a 20)
sum(count_over_time(
  {service="auth-service"} |= "LOGIN_FAILED" [1h]
))

# ACCOUNT_LOCKED nell'ultima ora (soglia arancio a 1)
sum(count_over_time(
  {service="auth-service"} |= "ACCOUNT_LOCKED" [1h]
))

# HMAC Failures su qualsiasi servizio (soglia rossa a 1)
sum(count_over_time(
  {service=~".+", level="WARN"} |= "HMAC" [1h]
))

# Trend eventi/min --grafico time-series
sum by (service) (rate(
  {service=~"auth-service|stay-service|billing-service"}
  |= "LOGIN_FAILED|CHECK_IN_FAILED|CHECK_OUT_FAILED" [5m]
))
```

Scelte progettuali rilevanti

- **Profilo Spring come guard**: il `<springProfile>` in `logback-spring.xml` assicura che l'appender Loki sia attivo esclusivamente quando il servizio viene avviato in Docker (`SPRING_PROFILES_ACTIVE=auth-service`). In ambiente di test JUnit non è presente alcun profilo attivo: solo il `ConsoleAppender` è attivo, nessuna connessione a Loki viene tentata.
- **AsyncAppender con `neverBlock=true`**: se Loki è irraggiungibile (es. container non ancora avviato), i log vengono scartati silenziosamente dalla coda senza mai bloccare i thread dell'applicazione. La resilienza funzionale dell'applicazione non dipende dalla disponibilità del sistema di logging.
- **Label Loki `service/level`**: il pattern di label `service=${appName},level=%level` permette query LogQL efficienti con filtro per servizio e livello di log, abilitando l'aggregazione cross-service dei log di audit.
- **Provisioning automatico Grafana**: datasource Loki e dashboard vengono caricati automaticamente all'avvio tramite i file YAML in `docker/grafana/provisioning/`, eliminando ogni configurazione manuale post-deploy.

Riferimento Commit: `feat(security): GAP-4 -- Loki log aggregation SIEM for audit events`

3.13 GAP-5 | T-GST-05 — GDPR Data Retention: Assenza di Policy sul Ciclo di Vita dei PII Ospite

Descrizione Il `guest-service` persiste in modo permanente i dati personali degli ospiti — nome, cognome, data di nascita, indirizzo, email, numero di telefono e, attraverso l'entità `IdentityDocument`, tipo documento, numero, date di emissione e scadenza, paese di rilascio. Queste informazioni vengono create al momento della prima prenotazione e rimangono nel database a tempo indeterminato: non esiste alcun meccanismo automatico di cancellazione, anonimizzazione o scadenza.

L'unica operazione di rimozione attualmente disponibile è il *soft-delete* (`active = false` via `@SQLDelete`), che nasconde il record dalle query applicative ma lascia i dati fisicamente presenti nel database. Questo comportamento è incompatibile con l'articolo 17 del GDPR (*diritto alla cancellazione*): un interessato che eserciti tale diritto non ottiene la cancellazione effettiva dei propri dati, bensì la loro semplice invisibilità a livello applicativo.

Inoltre non esiste alcun campo di consenso sull'entità `Guest` (`gdprConsentDate`, `consentPurpose`), rendendo impossibile dimostrare la base giuridica del trattamento in caso di ispezione da parte del Garante Privacy.

Classificazione OWASP **A04 — Insecure Design**: la mancanza di una policy di ritenzione è un difetto di progettazione del modello dei dati, non una vulnerabilità puntuale. Il rischio non dipende da un bug nel codice ma dall'assenza di un controllo progettato fin dall'inizio.

Contesto normativo italiano La situazione è resa più complessa dalla coesistenza di obblighi di legge che *impongono* la conservazione per periodi minimi e del diritto GDPR che ne chiede la cancellazione su richiesta:

Obbligo	Dato	Durata minima	Prevalenza
Circ. PS / D.Lgs. 196/2003	Dati alloggiati	5 anni	Su richiesta GDPR
Codice Civile art. 2220	Fatture / pagamenti	10 anni	Su richiesta GDPR
GDPR art. 17 par. 3 lett. b	—	Deroga per obblighi legali	Prevale sulla richiesta

In pratica: la richiesta di cancellazione di un ospite che abbia soggiornato negli ultimi 5 anni deve essere *rifiutata motivandola per legge*, non eseguita. Trascorso il periodo obbligatorio, i dati devono invece essere cancellati o anonimizzati.

Confronto con PMS professionali I sistemi PMS commerciali (Opera, Mews, Cloudbeds, Protel) adottano uniformemente il seguente modello:

1. **Retention period configurabile per hotel**: l'operatore imposta il periodo di conservazione (tipicamente 5–7 anni dall'ultimo soggiorno). Un job notturno identifica i profili scaduti e li *anonimizza*: il nome viene sostituito con un identificativo

pseudonimo, tutti i campi PII vengono azzerati e i documenti di identità vengono cancellati fisicamente. Lo storico contabile e gli stay associati vengono mantenuti per la rendicontazione fiscale.

2. **Hard-delete con guardia legale**: l'endpoint di cancellazione su richiesta dell'interessato verifica preventivamente l'esistenza di obblighi legali prevalenti (fatture nel periodo fiscale, dati alloggiati nel periodo PS). Se il blocco esiste, la richiesta viene rifiutata con un messaggio che indica la data a partire dalla quale la cancellazione sarà possibile. Altrimenti, PII e documenti vengono cancellati fisicamente e il profilo viene anonimizzato.

3. **Separazione fisica PII / dati operativi**: i dati di identità (entità `IdentityDocument`) sono mantenuti in una tabella separata dal profilo operativo (preferenze, storico soggiorni), consentendo retention differenziata: i dati di identità scadono prima, il profilo operativo (senza PII) può essere conservato più a lungo per analisi storiche e CRM.

L'architettura attuale del sistema presenta già una base favorevole: `IdentityDocument` è un'entità separata da `Guest` con `CascadeType.ALL`, il che rende tecnicamente semplice implementare una retention differenziata.

Soluzione implementata I quattro step del piano sono stati realizzati integralmente nel commit `56109eb`.

1. **Campo consenso — Flyway V4**: la migration `V4__add_gdpr_consent_date.sql` aggiunge la colonna `gdpr_consent_date DATE NOT NULL` alla tabella `guests`, con backfill automatico da `created_at` per i record esistenti (proxy legalmente difendibile del momento di registrazione in hotel). `GuestServiceImpl.createGuest()` stampa `LocalDate.now()` su ogni nuovo profilo.

2. **Retention job notturno**: `GuestRetentionJobServiceImpl` è annotato `@Scheduled(cron = "0 0 2 * * *")` e gira ogni notte alle 02:00. Usa il pre-filtro conservativo `FISCAL_MIN_YEARS = 10` anni per identificare i candidati senza Feign call: solo i profili la cui `gdprConsentDate` risale a più di 10 anni fa entrano nella pipeline. Per ciascun candidato verifica le due hold indipendenti tramite `StayServiceClient` e `BillingServiceClient`; in caso di indisponibilità del servizio il circuit-breaker ritorna `hasStays = true` (fail-closed: nessun ospite viene mai anonimizzato per errore). L'anonimizzazione imposta `firstName = "GDPR"`, `lastName = "ERASED_<id-prefix>"`, azzera tutti i campi PII, svuota `identityDocuments` (cascade hard-delete) e pone `active = false`.

3. **Hard-delete con guardia legale duale**: `GuestServiceImpl.deleteGuest()` esegue due check in sequenza:

   - **TULPS** — chiama `GET /api/v1/stays/guest/{id}/last-stay-date` (stay-service) e verifica che `lastStayDate + max(retentionYears, 5)` sia già trascorso.
   - **Fiscale** — chiama `GET /api/v1/invoices/guest/{id}/last-invoice-date` (billing-service) e verifica che `lastInvoiceDate + 10 anni` sia già trascorso.

Se almeno un vincolo è attivo, l'operazione è rifiutata con `HTTP 451 Unavailable For Legal Reasons` (RFC 7725) e un corpo `GdprLegalHoldResponse` contenente il campo `unlocksAt` (la data di sblocco più lontana) e `legalBasis` (`TULPS`, `FISCAL` o `TULPS_AND_FISCAL`). Il `GlobalExceptionHandler` intercetta `GdprLegalHoldException` e produce la risposta 451.

4. **Configurabilità per hotel — Flyway V5**: la migration `V5__add_guest_privacy_settings.sql` crea la tabella `guest_privacy_settings` (una riga per hotel).
   `GuestPrivacySettingsController` espone `GET/PUT /api/v1/guests/settings` con Jakarta Bean Validation (`@Min(5)`) che impedisce di scendere sotto il minimo TULPS. Il periodo di retention configurato viene rispettato sia dalla guardia legale hard-delete sia dal retention job.

Fix collaterale: FeignHeaderConfig mancante La `FeignHeaderConfig` del guest-service non calcolava la firma HMAC-SHA256 (`X-Internal-Signature`), a differenza di tutti gli altri servizi del progetto. Di conseguenza ogni chiamata Feign in uscita dal guest-service veniva rifiutata con HTTP 401 dagli `InternalAuthFilter` dei servizi downstream, e il circuit-breaker si attivava silenziosamente. Il file è stato allineato al pattern canonico (identico a stay-service) con l'aggiunta della dipendenza da `${internal.hmac.secret}` e del calcolo HMAC `username:role:hotelId` (T-GW-07).

Test Sono stati scritti 17 test suddivisi in tre classi, con copertura dei boundary value imposti dalla legge italiana:

- `GuestServiceImplDeleteGuardTest` (7 test): verifica i casi limite temporali — soggiorno a 4 anni e 364 giorni (bloccato), esattamente 5 anni (bloccato), 5 anni e 1 giorno (liberato); fattura a esattamente 10 anni (bloccata), 10 anni e 1 giorno (liberata); entrambi i vincoli attivi con `unlocksAt` corretto; retention configurabile per hotel a 7 anni.
- `GuestPrivacySettingsServiceImplTest` (5 test): get-or-create, default creation, update, update-when-none-exists, no-save-on-existing.
- `GuestRetentionJobServiceImplTest` (5 test): anonimizzazione completa, skip per TULPS hold, skip per fiscal hold, ospite senza storico, nessun candidato.

Stato **RISOLTO** — commit `56109eb`, fix warning `066f611`. Tutti e quattro gli step del piano di mitigazione sono implementati; la copertura del test include i boundary value temporali critici richiesti dalla normativa italiana.

3.14 GAP-6 — Nessun Controllo Preventivo sull'Isolamento Multi-Tenant

Descrizione A differenza degli altri GAP di questo capitolo, non è la scoperta di un singolo bug ma l'assenza di un *meccanismo preventivo*. T-BILL-04 e T-STAY-06 (Sezione "Broken Access Control — Isolamento Multi-Tenant", Capitolo 3) sono stati trovati manualmente durante l'esplorazione per la decisione multi-tenant (ADR-004) — nessun

controllo automatico impediva che una futura query repository su un'entità tenant-root (con colonna `hotel_id` propria) venisse scritta senza scoping `hotelId`. La scoperta di questa classe di bug dipendeva interamente dalla revisione umana, la stessa condizione che aveva già permesso a T-BILL-04 e T-STAY-06 di restare inosservati in produzione.

Mitigazione applicata `TenantIsolationArchTest` (ArchUnit) aggiunto ai 5 servizi con repository multi-tenant (auth, guest, frontdesk, billing, fb): fallisce la build se un metodo custom su un repository tenant-root non contiene `HotelId` nel nome e non è annotato `@TenantScopeExempt(reason=...)`.

TenantIsolationArchTest.java — struttura della regola

```java
@AnalyzeClasses(packages = "com.hotelpms.billing",
                importOptions = ImportOption.DoNotIncludeTests.class)
final class TenantIsolationArchTest {

    private static final Set<String> TENANT_ROOT_REPOSITORIES = Set.of(
            "com.hotelpms.billing.repository.InvoiceRepository",
            "com.hotelpms.billing.repository.InvoiceSequenceRepository");

    @ArchTest
    static final ArchRule CUSTOM_QUERY_METHODS_..._MUST_SCOPE_BY_HOTEL_ID =
        classes()
            .that(/* is one of TENANT_ROOT_REPOSITORIES */)
            .should(/* every declared method: name contains "HotelId"
                       OR is @TenantScopeExempt(reason=...) */);
}
```

I metodi ereditati da `JpaRepository`/`CrudRepository` (`findById`, `save`, `delete`) sono fuori scope — ArchUnit `getMethods()` restituisce solo i metodi dichiarati sull'interfaccia stessa; quei metodi sono già guardati al call-site (pattern `findByIdAndHotelId`, mai il `findById` ereditato, sulle entità tenant-root).

Eccezioni documentate 8 eccezioni verificate manualmente call-site per call-site (non semplicemente silenziate): login/registrazione pre-hotel-resolve su `UserAccount` (4, auth-service — lo username è l'unica credenziale disponibile prima che l'hotel del chiamante sia noto); batch GDPR notturno platform-wide su `Guest` (1, guest-service — pre-filtro, i vincoli di legal-hold per hotel sono verificati a valle); query di overlap prenotazioni con `roomIds` già scoped a monte da `verifyRoomsAvailability`/ `findCleanRooms` (3, frontdesk-service). billing-service e fb-service non necessitano eccezioni.

Esempio di eccezione documentata

```java
@TenantScopeExempt(reason = "Login lookup happens before the caller's "
    + "hotel is known -- the username is the only credential available "
    + "at this point in the auth flow. Usernames are globally unique "
    + "across the platform by design.")
Optional<UserAccount> findByUsername(String username);
```

Pulizia correlata Rimosso anche codice morto trovato nella stessa esplorazione, zero chiamanti in tutto il codebase: `Room.findByRoomNumberAndActiveTrue`, `Room.findAllByActiveTrue`, `Invoice.findByGuestId`, `Stay.findByReservationId`.

Verifica della regola Smoke test manuale: aggiunto temporaneamente un metodo non scoped a `InvoiceRepository`, confermato che `TenantIsolationArchTest` fallisce con un messaggio chiaro, poi rimosso. La regola cattura effettivamente la classe di bug per cui è stata scritta, non solo compila.

Gap residuo, noto e accettato `RoomType` non ha colonna `hotel_id` (catalogo tipologie camera oggi globale, non per-hotel) — richiede una migrazione di schema, non una query mancante; fuori scope per questa regola, documentato nel Javadoc di `TenantIsolationArchTest` e in `THREAT_MODEL.md` come gap da riaprire quando si affronta lo schema di `room_types`.

Riferimento Commit: `feat(security): ADR-004 -- ArchUnit rule enforcing hotel_id scoping`

Capitolo 4

Gestione Autenticazione e Autorizzazione

Questo capitolo presenta una visione sintetica e trasversale del sistema di identità e controllo degli accessi risultante dall'intero ciclo di hardening. I dettagli implementativi di ciascun intervento sono documentati nel Capitolo 3; qui l'obiettivo è offrire una lettura architetturale complessiva che mostri come i singoli interventi si raccordino in un modello coerente di difesa in profondità.

4.1 Ridisegno del flusso di autenticazione

Il flusso di autenticazione nella baseline era un unico JWT con scadenza di 24 ore, privo di revoca e senza binding al tenant dell'utente. Il ridisegno, completato in cinque iterazioni successive (commit `T-AUTH-02`, `T-AUTH-03`, `T-AUTH-03 upgrade Argon2id`, `T-AUTH-04`, `T-GW-06`, `T-AUTH-04 residuo`), produce il seguente flusso di autenticazione:

1. **Login** (`POST /api/v1/auth/login`): verifica lockout → verifica password (Argon2id via DelegatingPasswordEncoder, memory-hard) → genera *access token* (TTL 15 min, claim `role`, `hotelId`, `tv`) + *refresh token* (TTL 7 giorni, `jti` UUID, `typ=refresh`) → emette tre cookie: `jwt`, `refresh_token` (entrambi httpOnly/Secure/SameSite=Strict) e `csrf_token` (non-httpOnly, per il Double-Submit-Cookie). Scrive la chiave Redis `user:tv:<username>` con il token version corrente.

2. **Richiesta autenticata**: il browser invia automaticamente il cookie JWT. L'`AuthenticationFilter` del gateway valida la firma HMAC-SHA256 del JWT, estrae `username`, `role`, `hotelId`, inietta `X-Auth-User`, `X-Auth-Role`, `X-Auth-Hotel` e `X-Internal-Signature` sulla richiesta verso il microservizio. Il `CsrfFilter` (ordine `HIGHEST_PRECEDENCE+2`) verifica che l'header `X-CSRF-Token` corrisponda al cookie `csrf_token` per ogni metodo mutante.

3. **Refresh** (`POST /api/v1/auth/refresh`): valida il refresh token (firma, scadenza, tipo, JTI non blacklistato, token version Redis) → blacklista il JTI consumato → emette nuovo token pair con token version aggiornata.

4. **Cambio password** (`POST /api/v1/auth/change-password`): verifica la password corrente → incrementa `tokenVersion` in DB → sovrascrive Redis

`user:tv:<username>` → emette nuovo token pair (sessione corrente mantenuta) → ogni refresh token precedente viene rifiutato alla prossima rotazione (TV mismatch → 401).

5. **Logout** (`POST /api/v1/auth/logout`): blacklista il JTI del refresh token → azzera i tre cookie (`jwt`, `refresh_token`, `csrf_token`). Il log `[AUTH] LOGOUT | user=...` garantisce tracciabilità.

6. **Reset password admin** (`PATCH /api/v1/auth/users/{id}/reset-password`): ADMIN/OWNER invia nuova password → Argon2id encode → `mustChangePassword=true` → incrementa `tokenVersion` → sovrascrive Redis `user:tv:<username>` → 204 No Content. Tutte le sessioni attive dell'utente vengono invalidate (TV mismatch al prossimo refresh). Al login successivo il sistema reindirizza a `/profile` per impostare una nuova password personale. Commit: `G3 -- reset password con invalidazione sessioni (ADMIN/OWNER)`.

Fix inizializzazione hash admin — Flyway V6. L'adozione di `DelegatingPasswordEncoder` richiede che ogni hash in database sia prefissato con il nome dell'algoritmo (es. `{bcrypt}`) per permettere il dispatch corretto al verificatore. Il seed iniziale dell'utente `admin` in Flyway era privo di tale prefisso, causando un crash `IllegalArgumentException` al primo login. La migrazione `V6__fix_admin_password_bcrypt_prefix.sql` corregge l'hash con una guardia idempotente (`WHERE NOT password LIKE '\{bcrypt\}%'`) che risulta no-op su installazioni già corrette. Commit: `fix(security): V6 Flyway -- prefisso bcrypt mancante sull'hash admin iniziale`.

Limitazione operativa — recovery accesso amministrativo. In assenza di un secondo account ADMIN attivo, il ripristino dell'accesso amministrativo richiede accesso diretto al database PostgreSQL del container `auth-service` (es. `UPDATE user_account SET active = true WHERE ...` e reset dell'hash). Non esiste un endpoint applicativo di recovery: questa è una limitazione operativa nota, da documentare nel runbook di deployment.

4.2 Modello RBAC

Il sistema adotta un modello **Role-Based Access Control** a tre ruoli, con enforcement a tre livelli:

Ruolo	Accesso	Perimetro operativo
`ADMIN`	Totale	Tutte le funzionalità, tutte le risorse dell'hotel
`OWNER`	Totale	Identico ad ADMIN + dashboard reportistica
`RECEPTIONIST`	Limitato	Ospiti, prenotazioni, check-in/out, F&B; nessun accesso a fatture e dashboard finanziaria

Enforcement a tre livelli

1. **Gateway (stateless)**: l'`AuthenticationFilter` valida il JWT e inietta `X-Auth-Role`; le route del gateway non filtrano per ruolo (delegato al layer applicativo), ma rifiutano qualsiasi richiesta priva di JWT valido con HTTP 401.

2. **Microservizi (Spring Security)**: ciascun microservizio popola il `SecurityContext` tramite `InternalAuthFilter` con un `SimpleGrantedAuthority("ROLE_"+role)` derivato dall'header `X-Auth-Role`. I metodi di servizio e i controller usano `@PreAuthorize` o `hasRole()` di Spring Security per verificare il ruolo sul thread corrente. Un audit post-sviluppo ha identificato due endpoint (`POST /stays/reports/alloggiati/submit` e `PUT /stays/settings`) privi di `@PreAuthorize` lato server — l'accesso era limitato solo via UI. Entrambi sono stati corretti aggiungendo `@PreAuthorize("hasAnyRole('ADMIN','OWNER')")` con `@EnableMethodSecurity` abilitato su `SecurityConfig`. `AccessDeniedException` è gestito dal `GlobalExceptionHandler` che restituisce HTTP 403 anziché 500; il comportamento è verificato da `StayControllerSecurityTest` con test `@WebMvcTest` e HMAC reale. Commit: `fix(security): RBAC -- @PreAuthorize su StayController (submit/download alloggiati)` e `fix(security): RBAC -- @PreAuthorize su HotelSettingsController.updateSettings`.

3. **Frontend (React)**: il componente `ProtectedRoute` con prop `allowedRoles` redirige a `/` gli utenti non autorizzati prima di renderizzare la pagina (T-FE-03). Questo terzo livello è una misura *UX*; la garanzia di sicurezza effettiva è fornita dai livelli 1 e 2.

4.3 Gestione del ciclo di vita del JWT

Il JWT in produzione porta le seguenti claim obbligatorie:

Claim	Tipo	Scopo
`sub`	`String`	Username dell'utente autenticato
`role`	`String`	Ruolo (`ADMIN`, `OWNER`, `RECEPTIONIST`)
`hotelId`	`String` (UUID)	Tenant binding: iniettato come `X-Auth-Hotel`
`tv`	`int`	Token version: invalidazione bulk al cambio password
`iat`	`long`	Issued at (standard JWT)
`exp`	`long`	Expiration: 15 min (access), 7 giorni (refresh)
`jti`	`String` (UUID)	Solo refresh token: identificatore per blacklist Redis
`typ`	`String`	Solo refresh token: valore fisso `"refresh"`

Stato della revoca Il sistema gestisce due categorie di revoca:

- **Individuale** (logout, furto rilevato): il JTI del refresh token viene scritto in Redis con chiave `rt:blacklist:<jti>` e TTL pari alla vita residua del token. Scadenza automatica, nessun job di manutenzione.

- **Bulk** (cambio password): la chiave Redis `user:tv:<username>` viene sovrascritta con il nuovo valore di `tokenVersion`. Qualsiasi refresh token con `tv` inferiore viene rifiutato in fase di rotation, senza enumerare i JTI.

Firma HMAC inter-service Oltre alla firma JWT (HS256), ogni richiesta interna è autenticata da una firma HMAC-SHA256 calcolata su `"username:role:hotelId"` e trasmessa nell'header `X-Internal-Signature`. L'`InternalAuthFilter` di ogni microservizio verifica la firma con confronto `MessageDigest.isEqual` (constant-time) prima di popolare il `SecurityContext`, rendendo impossibile sia l'impersonation sia il tampering del tenant context (T-GW-01, T-GW-07).

4.4 Protezioni contro attacchi comuni all'autenticazione

La tabella seguente sintetizza le protezioni attive contro i principali vettori di attacco sull'autenticazione, con riferimento alla categoria OWASP e al commit di implementazione.

Vettore	Protezione	OWASP	Commit
Brute force credenziali	Lockout 5 tentativi / 15 min; rate limit gateway 5 req/s per IP	A07:2025	`a90e462`
User enumeration	Messaggio uniforme `INVALID_CREDENTIALS`	A07:2025	baseline
Password debole in DB	Argon2id memory-hard (DelegatingPasswordEncoder); BCrypt legacy migrati on login	A04:2025	`4e1f1e9`, `446ad8d`
Token theft (access)	TTL 15 min; cookie httpOnly/Secure/SameSite	A04:2025/ A07:2025	`5ce3010`
Token theft (refresh)	Rotation + blacklist JTI Redis; bulk revoca TV	A07:2025	`5ce3010`, `b473265`
Session hijacking CSRF	Double-Submit Cookie (header `X-CSRF-Token` == cookie)	A01:2025	`e146b17`
JWT forging	HS256 con segreto 48 byte random; fail-fast senza `JWT_SECRET`	A04:2025	`91d9484`, `6dc6ea4`
Header injection inter-service	HMAC-SHA256 su `username:role:hotelId` (constant-time compare)	A01:2025	baseline, `5bdc594`
Tenant hopping (IDOR)	`hotelId` nel JWT; X-Auth-Hotel iniettato; query hotel-scoped; 404 IDOR-safe	A01	`b469e21` + `T-GST/RES/BILL/FB`
XSS → token exfiltration	httpOnly cookie; ESLint ban `dangerouslySetInnerHTML`; CSP nginx	A05:2025	`3655906`, `b901a9f`

Postura complessiva Al termine del ciclo di hardening, il sistema implementa tutti i controlli minimi raccomandati dall'OWASP Authentication Cheat Sheet e dalla NIST SP 800-63B per applicazioni aziendali: password hashing con funzione lenta, lockout progressivo, revoca dei token, binding crittografico del tenant context, e audit logging di ogni evento di autenticazione verso un aggregatore centralizzato (Loki + Grafana). La superficie di attacco residua nella *nostra codebase* è ridotta a un singolo alert aperto nel Security tab di GitHub: **CVE-2026-42577** (Netty native epoll, rischio pratico nullo perché Spring Boot non attiva il trasporto native epoll in configurazione JDK NIO). Gli aggiornamenti a **Spring Boot 3.5.x** e **Spring Cloud 2025.0.x** hanno risolto CVE-2026-22739 (config-server), CVE-2025-41253 (gateway EL injection) e le CVE Netty della serie 4.1.13x. I 108 alert Trivy rilevati sulle immagini di terze parti (Grafana, postgres) sono stati analizzati, triaggiati e dismissi con motivazione tecnica documentata (si veda DEP-CVE-04).

Appendice A

Appendice A — Implementazioni Avanzate (AI-assisted)

Questa appendice raccoglie le implementazioni di sicurezza sviluppate con supporto diretto di **Claude Code** (Anthropic) che vanno oltre i requisiti minimi della traccia d'esame. Il codice è funzionante e integrato nel sistema; è documentato qui per completezza architetturale, non rivendicato come competenza autonoma esclusiva.

A.1 Autenticazione avanzata

T-AUTH-04 residual — Token versioning e revoca bulk (Redis) Al cambio password, il sistema incrementa `tokenVersion` in PostgreSQL e aggiorna la chiave Redis `user:tv:<username>`. Qualsiasi refresh token con `tv` inferiore al valore corrente è rifiutato alla prossima rotazione, invalidando tutte le sessioni precedenti. La finestra di esposizione residua è limitata ai soli access token in volo (massimo 15 minuti). Commit: `fix(security): GAP-1 -- password change revokes sessions via tokenVersion`

T-AUTH-05 — Audit log strutturati (OWASP A09) Ogni evento di autenticazione (login/fallimento/lockout/logout) produce una riga strutturata con prefisso `[AUTH]`, livello INFO/WARN, username e motivazione. Corrispondenza con OWASP A09:2025 — Security Logging & Alerting Failures. Commit: `fix(security): T-AUTH-05 -- structured auth audit log`

A.2 Sicurezza inter-service avanzata

T-GW-07 — HMAC-SHA256 inter-service con binding sul tenant Header `X-Internal-Signature` = HMAC-SHA256 di `"user:role:hotelId"`. Verificato con `MessageDigest.isEqual` (constant-time) da ogni microservizio. Impedisce impersonation e tampering del tenant context. Commit: `fix(security): T-GW-07 -- HMAC inter-service tenant binding`

T-GW-08 — Anti-replay (timestamp + nonce) su HMAC inter-service Il payload firmato T-GW-07 (`"user:role:hotelId"`) non includeva alcun elemento temporale: una firma catturata restava una credenziale valida indefinitamente, senza possibilità di revoca se non ruotando il segreto condiviso.

Esteso a `"user:role:hotelId:timestamp:nonce"`: l'API Gateway genera timestamp (epoch millis) e nonce (UUID) casuali ad ogni richiesta, propagati come `X-Auth-Timestamp`/`X-Auth-Nonce`. Ogni `InternalAuthFilter` verifica una finestra di tolleranza di 60s sul timestamp e reclama il nonce su un `NonceStore` backed da Redis (`SETNX`, TTL 120s) — un secondo reclamo dello stesso nonce entro la finestra viene rifiutato come replay. Commit: `feat(security): T-GW-08 -- timestamp+nonce anti-replay on internal HMAC auth`

T-GW-09 — Seconda barriera gateway mancante su una nuova risorsa admin-only Ogni risorsa la cui scrittura è riservata ad ADMIN/OWNER (es. `/api/v1/room-types`) è protetta da due barriere indipendenti: il controller applica `@PreAuthorize("hasAnyRole('ADMIN','OWNER')")`, e l'API Gateway rifiuta a monte qualunque scrittura verso quel prefisso per un ruolo diverso, tramite `WRITE_RESTRICTED_PREFIXES` in `AuthenticationFilter`. La nuova risorsa `/api/v1/rate-calendar` (bulk-apply prezzi su più tipologie camera) è nata con la sola barriera a livello controller: non uno sfruttabile isolato (`RECEPTIONIST` resta comunque bloccato dal `@PreAuthorize`), ma una rottura del pattern a due livelli che rende l'enforcement dipendente da un solo punto di fallimento invece di due. Aggiunto `"/api/v1/rate-calendar"` a `WRITE_RESTRICTED_PREFIXES`, allineando la risorsa al resto delle scritture admin-only; la lettura (`GET`) resta aperta a tutti i ruoli operativi, invariata. Commit: `fix(security): T-GW-09 -- gateway-level RBAC on /api/v1/rate-calendar bulk-apply`

T-AUTH-10 — AccessDeniedException mai gestita in auth-service, 500 invece di 403 `UserManagementController` applica `@PreAuthorize`, ma il `GlobalExceptionHandler` di auth-service non aveva mai un `@ExceptionHandler` esplicito per `AccessDeniedException`: il diniego cadeva nel catch-all generico `Exception.class` e tornava HTTP 500 invece di 403. Non uno sfruttabile isolato (`@PreAuthorize` ha sempre negato correttamente l'accesso), ma un client o un sistema di monitoring non può distinguere un diniego di autorizzazione legittimo da un guasto server reale, e lo status HTTP standard viene nascosto dietro un errore generico. Trovato analizzando un finding OSINT non correlato (endpoint Actuator che tornava 500 invece di 404): la causa comune era un catch-all troppo largo, duplicato in 5 dei 6 microservizi invece di vivere nella classe condivisa `AbstractProblemDetailAdvice` (`common-web-lib`). Corretto completando la migrazione di auth-service, frontdesk-service e notification-service a estendere quella classe condivisa invece di duplicare gli handler localmente — `AccessDeniedException` ora eredita l'handler già corretto usato da billing/fb/guest-service, senza codice nuovo in auth-service. Aggiunti alla base condivisa anche quattro handler per eccezioni Spring native mai gestite esplicitamente in nessun servizio (`NoResourceFoundException`→404, `MethodArgumentTypeMismatchException`→400, `HttpRequestMethodNotSupportedException`→405, `HttpMediaTypeNotSupportedException`→415). Commit: `fix(security): T-AUTH-10 -- auth-service inherits AccessDeniedException(403) handling`

GAP-11 — XXE hardening mancante su FatturaPAServiceImpl Uno scan Semgrep mirato su injection/XXE (595 regole Java+OWASP su 905 file) ha segnalato `DocumentBuilderFactory` e `TransformerFactory` in `FatturaPAServiceImpl` (billing-service, generazione XML FatturaPA) privi di hardening DOCTYPE/entità esterne. Non

sfruttabile allo stato attuale: il factory costruisce solo un `Document` vuoto in memoria, mai un parsing di input esterno; il transformer serializza quel DOM, mai uno stylesheet esterno. Corretto comunque come difesa in profondità, stesso standard già applicato in `AlloggiatiWebSenderServiceImpl` (T-STAY-03): `disallow-doctype-decl` sul factory, `ACCESS_EXTERNAL_DTD/ACCESS_EXTERNAL_STYLESHEET` vuoti sul transformer. Zero cambio di comportamento osservabile. Commit: `fix(security): GAP-11 -- XXE hardening on FatturaPAServiceImpl`

GAP-12 — PropertyReferenceException mai gestita, 500 invece di 400 Lo stesso audit Semgrep ha portato a verificare manualmente ogni endpoint paginato: un `sort` su una property inesistente (es. `?sort=nonEsiste`) fa fallire la validazione di Spring Data contro il metamodello JPA *prima* che venga generato SQL — non è injection — ma nessuno dei 6 `GlobalExceptionHandler` gestiva `PropertyReferenceException`, che cadeva nel catch-all generico e tornava 500. Stesso pattern di T-AUTH-10 (eccezione client-side mascherata da errore server). Fix: nuovo handler condiviso in `AbstractProblemDetailAdvice`, ereditato automaticamente da tutti e 6 i servizi senza modifiche per-servizio. Commit: `fix(security): GAP-12 -- PropertyReferenceException(400) shared handler`

GAP-13 — Actuator fb-service esposto per intero in config locale `fb-service/application-fb-service.yml` (config locale, non quella servita da config-service) esponeva l'intero Actuator (`include: "*"`, incluso `/actuator/env` e `/actuator/heapdump`) mentre config-service serve già per lo stesso servizio il valore corretto (`prometheus,health,info`, identico agli altri 6 servizi) — residuo locale mai allineato, unico dei 4 file `application-<service>.yml` a toccare la config actuator. Trovato dalla regola Semgrep `spring-actuator-fully-enabled-yaml`. Fix: allineato al valore già corretto in config-service. Commit: `fix(security): GAP-13 -- fb-service local actuator exposure aligned`

GAP-14 — nginx gira come root nell'immagine frontend `frontend/Dockerfile` non dichiarava alcun `USER`: il processo master nginx girava come root (immagine base `nginx:alpine` standard). Fix: swap a `nginxinc/nginx-unprivileged:alpine` (immagine ufficiale non-root, ascolto interno su 8080 — le porte <1024 richiedono root per il bind), `USER nginx` esplicito, `nginx.conf` aggiornato (`listen 8080`), `docker-compose.yml` porte `"80:8080"`. La porta host pubblicata resta 80 — nessun impatto esterno o sulla CI. Verificato dal vivo: rebuild, container healthy, `curl` dall'host su :80 risponde 200 come prima del cambio. Commit: `fix(security): GAP-14 -- frontend nginx runs as non-root (nginx-unprivileged)`

GAP-15 — GitHub Actions non pinnate a SHA, Dependabot senza cooldown `.github/workflows/ci.yml` referenziava 7 Action per tag mutabile (`actions/checkout@v7`, `setup-java@v5`, `cache@v6`, `upload-artifact@v7`, `setup-node@v7`, `download-artifact@v8`, `codeql-action/upload-sarif@v4`) — un tag può essere ripuntato dal maintainer senza preavviso (supply-chain; incidente reale di riferimento: `tj-actions/changed-files`, 2025). Il progetto pinnava già a SHA `aquasecurity/trivy-action` e `docker/setup-buildx-action`; le altre 7 erano rimaste sul pattern precedente. `.github/dependabot.yml` non impostava alcun `cooldown` sugli 11 blocchi `package-ecosystem` — una dipendenza appena pubblicata (potenzialmente

compromessa) veniva proposta in PR immediatamente. Fix: le 7 Action pinnate a SHA (stesso pattern `@<sha40> # vX.Y.Z` già in uso, SHA risolti dal tag corrente — zero cambio di comportamento); `cooldown: default-days: 7` aggiunto a tutti gli 11 blocchi Dependabot. Commit: `fix(security): GAP-15 -- pin GitHub Actions to SHA, add Dependabot cooldown`

GAP-16 — postgres gira come root: rischio accettato, non una fix Stessa regola Semgrep di GAP-14 segnala anche `docker/postgres/Dockerfile` privo di `USER`. A differenza di nginx, qui la fix "ovvia" romperebbe funzionalità esistente: `entrypoint-wrapper.sh` scrive `/etc/pgbackrest/pgbackrest.conf` e ne fa `chown postgres:postgres` (un processo non-root non può cambiare proprietario a un altro utente), poi ogni comando verso Postgres/pgbackrest passa già per `gosu postgres ...` — esattamente il pattern privilege-drop dell'immagine ufficiale `postgres:15-alpine` upstream (il suo stesso `docker-entrypoint.sh`, quando parte come root, fa `gosu postgres` internamente prima di avviare il server; Postgres stesso rifiuta di girare come root). Non esiste un'immagine "unprivileged" equivalente: pgBackRest richiede root transitorio per sistemare i permessi di un volume montato al primo avvio. Nessuna modifica di codice — rischio residuo accettato, stesso trattamento di T-OPS-02.

T-NOTIF-01 — notification-service mai censito nel modello delle minacce notification-service (non instradato dal gateway, raggiungibile solo dalla rete Docker interna, 3 endpoint solo-scrittura che innescano invio email transazionali contenenti PII ospite nel corpo/subject) era già protetto dallo stesso `InternalAuthFilter` HMAC+anti-replay di T-GW-08 fin dalla sua introduzione, ma non compariva mai in `THREAT_MODEL.md` se non come voce della tabella asset — zero threat, zero mitigazione tracciati. Senza quella protezione, un chiamante privo di firma HMAC valida ma con accesso alla rete interna avrebbe potuto innescare invii email arbitrari (subject/corpo/destinatario a scelta del chiamante) tramite l'infrastruttura email legittima dell'hotel — un vettore di phishing verso gli ospiti. Formalizzato durante l'estrazione di `internal-auth-lib` (GAP-7), che ha portato a riesaminare insieme la protezione di tutti e 6 i servizi. Commit: `test(security): characterization tests for InternalAuthFilter on auth/notification-service`

GAP-7 — Sei copie divergenti di InternalAuthFilter, un solo punto di verità mancante `InternalAuthFilter/NonceStore/RedisNonceStore` esistevano come 6 copie quasi-identiche (auth/billing/fb/frontdesk/guest/notification-service). 5 erano byte-identiche per disciplina nel copia-incolla, non per un meccanismo che lo garantisse — nessun controllo impediva a una futura modifica di introdurne una sesta divergenza silenziosa. auth-service era la prova che il rischio non era ipotetico: allowlist invertita anziché denylist, controllo presenza header con `== null` anziché `StringUtils.hasText` (un header `X-Auth-User: ""` non veniva rifiutato), contratto 401 diverso (`sendError`+codice stringa anziché corpo JSON), zero logging sui rigetti. Tre di queste sono difetti di sicurezza reali, non solo stilistici, scoperti da questo lavoro di dedup e non da un pentest mirato. Fix in tre passi: test di caratterizzazione per fissare il comportamento reale di ciascun servizio prima di toccare nulla; riconciliazione di auth-service sul contratto condiviso; estrazione in un nuovo modulo Gradle `internal-auth-lib` (libreria pura, nessuna autoconfigurazione Spring) consumato da tutti e 6 i servizi tramite `new InternalAuthFilter(hmacSecret, nonceStore, excludedPathPrefixes)` nel rispettivo `SecurityConfig`, altrimenti invariato. Verificato

dal vivo su tutti e 6 i servizi (richiesta non firmata→401, timestamp scaduto→401, nonce riusato→401, `/actuator/health/liveness` senza header→200) più un flusso applicativo autenticato completo via browser attraverso il gateway reale — nessuna regressione. Una revisione di sicurezza indipendente (subagent dedicato) pre-merge ha confermato l'assenza di vulnerabilità sfruttabili e suggerito un solo indurimento difensivo aggiuntivo, poi applicato: il matching dei prefissi esenti da HMAC (`/actuator`) avveniva per sottostringa grezza anziché per confine di segmento — non sfruttabile nella configurazione attuale (Spring Security richiede comunque autenticazione su qualunque path non esplicitamente permesso), ma corretto comunque per eliminare il rischio latente su futuri path esenti. Commit: `test(security): characterization tests`, `fix(security): reconcile auth-service InternalAuthFilter onto the shared contract`, `refactor(security): extract internal-auth-lib`, `fix(security): match excludedPathPrefixes on path-segment boundaries`

T-FB-03 — `@PreAuthorize` mai applicato su MenuItemController `SecurityConfig` di fb-service mancava l'annotazione `@EnableMethodSecurity`: i tre `@PreAuthorize("hasAnyRole('ADM` su `MenuItemController` (create/update/delete) non venivano mai valutati da Spring — qualunque ruolo autenticato, incluso RECEPTIONIST, poteva creare, modificare o cancellare voci di menu. Il gap non è stato trovato da un pentest ma diffando le 5 copie di `SecurityConfig` prima di consolidarle (GAP-8 sotto): l'assenza dell'annotazione era invisibile leggendo il singolo file, evidente solo confrontandolo con gli altri 4 servizi — lo stesso schema di scoperta di GAP-7. Il test unitario esistente (`MenuItemControllerTest`) non l'aveva mai intercettato perché usa `MockMvcBuilders.standaloneSetup()`, che non carica la security filter chain reale. Fix: `@EnableMethodSecurity` aggiunta a fb-service; applicata anche a guest-service (che non ha ancora nessun `@PreAuthorize`) come difesa in profondità, cosicché questa classe di bug non possa ripresentarsi silenziosamente in futuro. Nuovo `MenuItemControllerSecurityTest` (`@WebMvcTest` con la `SecurityConfig` reale importata, non uno standalone MockMvc) come regressione mirata. Commit: `fix(security): shared internal API security filter chain + missing RBAC enforcement`

GAP-8 — FeignHeaderConfig, SecurityConfig e GlobalExceptionHandler: stesso pattern di GAP-7 `FeignHeaderConfig` (4 copie, 107–125 righe) e `SecurityConfig` (5 copie, 75–79 righe) presentavano lo stesso rischio strutturale già osservato in GAP-7: duplicazione mantenuta identica per disciplina nel copia-incolla, non per un meccanismo che lo garantisse. Diffando le copie prima di consolidarle sono emersi due difetti reali, non solo differenze cosmetiche di javadoc: (1) T-FB-03 sopra; (2) né fb-service né billing-service gestivano `AccessDeniedException` nel proprio `GlobalExceptionHandler` — un diniego `@PreAuthorize` (corretto: l'autorizzazione era comunque negata, non un bypass) cadeva nell'handler generico e tornava HTTP 500 anziché 403, un contratto di risposta scorretto che billing-service aveva da quando `OwnerReportController` ha adottato `@PreAuthorize`, mai emerso perché il traffico reale non aveva mai esercitato il path di diniego in modo osservabile.

Fix: `FeignHeaderConfig` estratto in `InternalFeignAuthInterceptor` (`internal-auth-lib`) — l'unica divergenza comportamentale reale, il fallback di guest-service per il job batch GDPR che gira fuori da un contesto HTTP, è stata modellata come strategia (`FeignAuthFallbackProvider`) invece di un `if` duplicato in ogni copia. `SecurityConfig` estratto in `InternalApiSecurityFilterChainFactory`, stessa libreria. Handler

`AccessDeniedException` aggiunto a fb-service e billing-service (pattern già presente in frontdesk-service).

Nello stesso lavoro, anche le 6 copie di `GlobalExceptionHandler`: solo billing/fb/guest erano davvero identici (stesso host URI, stessa forma degli errori di validazione) e sono stati consolidati in `AbstractProblemDetailAdvice` (nuovo modulo `common-web-lib`). auth-service e frontdesk-service usano una forma diversa per gli errori di validazione (mappa campo→messaggio, non lista) e frontdesk-service genera un `traceId` di correlazione che gli altri non hanno — forzarli nella stessa base avrebbe significato perdere il traceId o cambiare silenziosamente il contratto di risposta, quindi sono rimasti separati. Trovati invece cinque host URI diversi per il campo `type` RFC 7807 (`hotel-pms.com`, `hotelpms.com`, `api.hotelpms.com`, `hotelpms.local/problems`) — nessuno è un dominio registrato (progetto pre-lancio, vedi `backup/DECISIONS.md`), nessun frontend o test e2e dipende dal valore esatto: normalizzati tutti su `https://hotel-pms.com/errors/`.

Verificato con build completa (`./gradlew clean build`, 193 task, zero warning PMD/checkstyle/SpotBugs) più i test di regressione esistenti su ciascun servizio toccato. Commit: `refactor(security): single HMAC Feign header config in internal-auth-lib`, `fix(security): shared internal API security filter chain + missing RBAC enforcement`, `refactor: shared RFC 7807 problem detail advice, normalize error-type host`

GAP-9 — Falso positivo CodeQL: CSRF "disabilitato" sulla filter chain interna CodeQL (regola `java/spring-disabled-csrf-protection`, severità HIGH) segnalava `csrf(AbstractHttpConfigurer::disable)` in `InternalApiSecurityFilterChainFactory` (`internal-auth-lib`, condivisa da billing/fb/frontdesk/guest/notification-service, estratta in GAP-8) come protezione CSRF disabilitata. Analisi confermata come falso positivo, non un fix applicativo: questa filter chain protegge esclusivamente chiamate interne service-to-service dietro il gateway, mai raggiungibili da un browser. La protezione CSRF esiste per contrastare richieste automaticamente corredate di credenziali (tipicamente cookie di sessione) inviate da un sito malevolo per conto di un utente autenticato — qui non si applica su due assi indipendenti: `sessionCreationPolicy(STATELESS)` non emette mai un cookie di sessione, e l'autenticazione avviene tramite firma HMAC (`InternalAuthFilter`, stesso meccanismo di T-GW-08) che una pagina web attaccante non può calcolare senza il segreto condiviso — esattamente il caso che la stessa documentazione Spring esclude esplicitamente ("only for services used only by non-browser clients"). Il traffico browser-facing reale ha una protezione CSRF distinta e corretta sul gateway (`CsrfFilter`, T-GW-05, double-submit cookie).

Verificato con revisione indipendente (agente `security-auditor`, non solo lettura del commento di soppressione): confermato che nessuno dei 5 servizi espone questi endpoint direttamente a un browser (tutti raggiungibili solo attraverso l'API Gateway) e che non esiste alcuna credenziale auto-allegata sfruttabile su questo percorso. Aggiunto un commento di soppressione `codeql[...]` inline sopra la chiamata — stesso pattern già usato per il falso positivo log-injection di GAP-7 — e alert dismissato su GitHub con motivazione. Commit: `docs(security): GAP-9 -- suppress CodeQL false-positive CSRF alert on internal-only filter chain`

T-OPS-01 — RPO fisso a 24h: da dump pianificato a WAL archiving continuo Il meccanismo di backup originale (`docker/db-backup/`, `pg_dumpall` giornaliero, gzip, cifratura `age`, upload via `mc`) aveva un RPO fisso di 24h: un disastro poco prima

del prossimo dump pianificato perdeva fino a un'intera giornata di lavoro. Una decisione precedente (`backup/DECISIONS.md §3.5`, 2026-07-30) aveva accettato esplicitamente questo RPO, rifiutando PITR/WAL archiving per complessità non giustificata dal profilo di rischio del prodotto. La decisione è stata riaperta e ribaltata il 2026-08-01 su richiesta diretta dell'utente — non da uno dei quattro criteri di revisione originariamente documentati, ma dal ragionamento sul caso peggiore: un dump a mezzanotte con una perdita dati poco prima significa perdere quasi un giorno di lavoro.

Fix: sostituito interamente `docker/db-backup/` con **pgBackRest**, integrato nel container `postgres` stesso (`docker/postgres/`) — l'`archive_command` deve girare come sottoprocesso del motore Postgres, non in un container satellite. WAL archiving continuo (`archive_timeout=120s`, RPO in minuti, non più 24h) più backup full settimanali/incrementali giornalieri su due repository pgBackRest indipendenti: `repo1` locale e `repo2` off-site (Backblaze B2, stesso bucket già provisionato).

Bug trovato e corretto durante la verifica dal vivo, non ipotetico: `pgbackrest backup` opera su un repository per invocazione (default `repo1` se `-repo` non specificato), a differenza di `archive-push` che scrive automaticamente su tutti i repository configurati. La prima versione di `backup-scheduler.sh` non specificava mai `-repo`: il WAL raggiungeva correttamente entrambi i repository, ma i backup full/incrementali finivano solo su `repo1` — `repo2` aveva WAL continuo ma zero backup validi (`pgbackrest info` riportava esplicitamente `repo2: error (no valid backups)`). Fix: lo scheduler itera ora esplicitamente su ogni repository configurato.

Drill PITR reale eseguito e verificato, non solo scritto: creata una tabella marker, inserita una riga, forzato uno switch WAL (`pg_switch_wal()`), eseguito `pgbackrest restore -type=time -target=<timestamp prima dell'insert>` in una directory scratch separata dallo stack live, avviata come istanza Postgres temporanea su una porta alternativa. La riga marker risulta assente nel restore, la tabella (creata prima del target) è presente, `flyway_schema_history` è coerente. Il log di pgBackRest conferma esplicitamente: *"recovery stopping before commit of transaction ..., time 2026-08-01 18:48:18"* — esattamente la transazione dell'insert, fermata prima del commit. Verifica visiva sul bucket B2 reale (`secure.backblaze.com`) richiede il login dell'utente al proprio account — non eseguibile da un agente automatico senza le sue credenziali, lasciata come passo manuale esplicito.

Nuovo workflow `.github/workflows/backup-restore-drill.yml`: drill off-site settimanale (+ `workflow_dispatch`) su un runner GitHub effimero, non ancora eseguito dal vivo — richiede prima la configurazione dei secret CI. Commit: `feat(backup): continuous WAL archiving + PITR via pgBackRest, replaces daily dump`, `fix: include docker-compose.yml and remaining backup-hardening changes`

T-OPS-02 — Cambio del modello di cifratura backup: da asimmetrico a simmetrico Il WAL archiving continuo di T-OPS-01 richiede che pgBackRest scriva WAL cifrato in tempo reale, il che richiede la passphrase di cifratura presente sullo stesso host che fa girare Postgres. pgBackRest cifra con una passphrase **simmetrica** (`PGBACKREST_CIPHER_PASS`, `repo-cipher-pass`), non con una coppia di chiavi asimmetriche come lo schema `age` precedente. La proprietà che aveva lo schema `age` — solo la chiave pubblica sull'host di produzione, la privata mai — **non è replicabile** con questo strumento: se l'host di produzione viene compromesso, chi lo compromette ha anche la chiave per decifrare i backup passati e futuri.

Questo non è un rischio scoperto per caso ma un trade-off dichiarato esplicitamente all'utente *prima* dell'implementazione, in cambio di: WAL archiving continuo (T-OPS-01), verify/restore-check integrati in pgBackRest, meno codice infrastrutturale hand-rolled da mantenere (strumento maturo e mantenuto invece di script bash a mano, coerente con `ADR-002`). Documentato per iscritto in `backup/DECISIONS.md §3.5b` prima di procedere, non dopo. Commit: `feat(backup): continuous WAL archiving + PITR via pgBackRest, replaces daily dump`

T-GW-06 — Propagazione X-Auth-Hotel end-to-end Aggiunto campo `hotelId` a `UserAccount` e claim JWT corrispondente. L'API Gateway inietta `X-Auth-Hotel` su ogni richiesta a valle. Senza questo intervento tutti i servizi multi-tenant restituivano HTTP 401. Commit: `fix(security): T-GW-06 -- X-Auth-Hotel propagation end-to-end`

T-GW-02 — Rate limiting Redis token bucket proxy-aware Key resolver aggiornato per leggere `X-Forwarded-For` anziché l'IP TCP (dietro un proxy tutti i client condividerebbero un unico bucket). Rate limiting esteso a tutte le route business, non solo `/auth/**`. Commit: `fix(security): T-GW-02 -- proxy-aware rate limiting all routes`

T-GST-07 — `@PreAuthorize` mai applicato su GuestController e GuestPrivacySettingsController Trovato dal secondo giro di test esplorativo dal vivo (round 1 finding #7 + round 2 bug #6), stesso pattern di T-FB-03: nessuna `@PreAuthorize` né a livello classe né metodo su `GuestController` (delete/ anonimizzazione, export Art. 20 GDPR) né su `GuestPrivacySettingsController` (periodo di retention GDPR dell'hotel). L'unica protezione era la whitelist per-prefisso nell'api-gateway, che autorizza qualunque ruolo operativo (incluso RECEPTIONIST) su `/api/v1/guests/**`. Riprodotto dal vivo: un RECEPTIONIST cancellava/anonimizzava un ospite, esportava il dump PII completo e modificava la retention policy dell'hotel, tutte operazioni di primo livello mai pensate per quel ruolo. Fix: `@PreAuthorize("hasAnyRole('ADMIN','OWNER')")` su `deleteGuest/exportGuestData` in `GuestController` e a livello classe su `GuestPrivacySettingsController` — CRUD/ricerca/allegati documento restano invece aperti a RECEPTIONIST, operatività quotidiana del front desk. Nuovi `GuestControllerSecurityTest/GuestPrivacySettingsControllerSecurityTest` (`@WebMvcTest` con `SecurityConfig` reale, stesso pattern di `MenuItemControllerSecurityTest`). Verificato dal vivo: RECEPTIONIST → `403` sui 4 endpoint ristretti, `200` invariato su lettura. Commit: `fix(security): T-GST-07/T-BILL-07 -- missing RBAC on guest/invoice controllers`

T-BILL-07 — `@PreAuthorize` mai applicato su InvoiceController Stesso pattern di T-GST-07, trovato dal round 2 bug #5: nessuna `@PreAuthorize` sulle operazioni fiscalmente sensibili di `InvoiceController`. Riprodotto dal vivo con un RECEPTIONIST: cambio `documentType` FATTURA↔RICEVUTA, generazione (e blocco permanente) di un export FatturaPA, impostazione dello stato di trasmissione SDI, e lancio dell'export batch per il commercialista, tutte riuscite senza alcun controllo oltre l'autenticazione. Fix: `@PreAuthorize("hasAnyRole('ADMIN','OWNER')")` sui quattro endpoint (`document-type`, `fatturaPA`, `sdi-status`, `export` batch). Lasciati aperti a RECEPTIONIST: lettura/ricerca fatture, download PDF di cortesia, e i tre endpoint di chiama-

ta interservizio (`createInvoiceForStay`, `addCharge`, `getLastInvoiceDateForGuest`) — questi ultimi propagano via HMAC il ruolo dell'utente che ha originato l'azione reale (check-in, conferma ordine F&B, cancellazione ospite: quest'ultimo è proprio il chiamante del fix T-GST-06 sopra), non l'operatore che chiama `InvoiceController` direttamente — restringerli a ADMIN/OWNER avrebbe rotto il check-in di un RECEPTIONIST. Nuovo `InvoiceControllerSecurityTest`. Verificato dal vivo: RECEPTIONIST → `403` sui 4 endpoint, `200` invariato su lettura; ADMIN invariato (un tentativo di `document-type` su fattura `PAID` resta `409` per un guard di business preesistente, non `403` — nessuna regressione introdotta da questo fix). Commit: `fix(security): T-GST-07/T-BILL-07 -- missing RBAC on guest/invoice controllers`

GAP-17 — Trust boundary IP rotto dalla topologia reale nginx→gateway Trovato durante la ri-verifica post-hardening del 2026-08-16 da due sotto-agenti indipendenti (categorie *auth-jwt* e *access-control*), che hanno segnalato separatamente lo stesso problema su T-AUTH-12 e T-GW-10: entrambi i fix avevano sostituito la fonte di trust da un header spoofabile (`X-Forwarded-For`) al TCP `remoteAddress` grezzo — corretto in isolamento, ma nella topologia reale il browser non parla mai direttamente con `api-gateway`: passa sempre per `frontend` (nginx), `proxy_pass http://api-gateway:8080` sulla stessa rete Docker (`dmz-network`), senza alcuna configurazione di trust sugli header forwarded lato gateway. Ogni richiesta dal front-door standard arrivava quindi al gateway con lo stesso peer TCP — l'IP interno del container nginx, identico per *ogni* utente reale — collassando la chiave di lockout `(username, clientIp)` di T-AUTH-12 a `(username)` esattamente come prima del fix, e raggruppando tutti i client reali in un unico bucket di rate limiting condiviso (T-GW-10). Exploit concreto: 5 richieste `POST /api/v1/auth/login` con `username=admin` e password errata, dalla normale pagina di login pubblica, bloccavano il vero `admin` per 15 minuti, ripetibile indefinitamente.

Fix: `docker-compose.prod.yml` non pubblica più `api-gateway:8080` sull'host — solo `frontend:80` resta raggiungibile dall'esterno, verificato che `dmz-network` interno (condiviso da frontend e api-gateway) non dipende dal port publish tramite `docker compose config`. Nuovo flag `gateway.security.trust-forwarded-header` (`config-service`: `false` in `api-gateway.yml`, `true` solo in `api-gateway-prod.yml`) fa sì che `ClientIpFilter` si fidi dell'header `X-Real-IP` di nginx (sempre impostato da `$remote_addr`, overwrite mai append) solo quando quel flag è vero — cioè solo quando la porta diretta è effettivamente chiusa. `RateLimiterConfig` non ri-deriva più l'IP autonomamente: legge lo stesso header `X-Client-IP` già risolto da `ClientIpFilter`, eliminando la duplicazione di logica che aveva causato la regressione T-GW-10 in primo luogo.

Regressione trovata nello stesso giro di verifica: la prima versione del fix impostava il flag a `true` in `api-gateway-prod.yml`, ma nessun file del repository settava mai `SPRING_PROFILES_ACTIVE=prod` per `api-gateway` — unico servizio senza, a differenza degli altri 6 (che impostano `SPRING_PROFILES_ACTIVE=<nome-servizio>`). Spring Cloud Config non serve mai un file di profilo senza quel profilo attivo sul client: il flag restava `false` anche a porta chiusa, e GAP-17 restava di fatto aperto nonostante il fix. Aggiunto `SPRING_PROFILES_ACTIVE=prod` all'`environment` di `api-gateway` in `docker-compose.prod.yml`, verificato che il merge è additivo (tutte le variabili base restano) via `docker compose config`. Verificato anche: in dev (porta ancora aperta, flag `false`) un `X-Real-IP` o `X-Forwarded-For` forgiato inviato direttamente a `:8080` resta ignorato — finding #3 (security-report.md) non riapre in quell'ambiente. Commit:

`fix(security): trust nginx X-Real-IP at gateway only once api-gateway:8080 is closed (GAP-17)`

GAP-18 — Version-check assente sul soft-delete di Quotation e RateSeason Trovato durante la stessa ri-verifica post-hardening (sotto-agente *business-logic*): l'override `@SQLDelete` di `Quotation` e `RateSeason` (`UPDATE ... SET active = false WHERE id = ?`) non verificava né incrementava la colonna `version` aggiunta da T-QUOT-01/T-PRICE-01, a differenza di `Reservation` nello stesso codebase (`UPDATE ... SET active = false, version = version + 1 WHERE id = ? AND version = ?`). Poiché l'`UPDATE` generato da Hibernate per un `save()` normale scrive comunque tutte le colonne mappate, incluso `active`, una race `UPDATE`-vs-`DELETE` concorrente sulla stessa riga non generava mai un conflitto: un `DELETE` (staff B) poteva essere silenziosamente annullato da un `save()` concorrente (staff A) ignaro della cancellazione, "resuscitando" un preventivo appena cancellato come `ACCEPTED` con una prenotazione già creata, o riportando in vita una stagione tariffaria che un secondo admin credeva rimossa.

Fix: entrambi gli `@SQLDelete` allineati esattamente al pattern già in uso su `Reservation.java`.

Regressione trovata nella riverifica dello stesso fix: il pattern replicato da `Reservation` non enforcava mai davvero il controllo versione a runtime — `@SQLDelete` di Hibernate usa `ResultCheckStyle.NONE` di default, quindi una `DELETE` con `version` disallineata aggiorna zero righe e Hibernate non se ne accorge, riportando successo pur non avendo cancellato nulla (race inversa: una `DELETE` stale sopravvive silenziosamente a un `save()` concorrente che ha vinto per primo). Lo stesso difetto latente era già presente in `Reservation.java` fin da T-RES-01, non introdotto da questo fix. Aggiunto `check = ResultCheckStyle.COUNT` a tutti e tre gli `@SQLDelete` (`Quotation`, `RateSeason` e `Reservation`) — una `DELETE` a zero righe solleva ora la `StaleStateException` che `GlobalExceptionHandler` già mappa a `409 CONCURRENT_MODIFICATION`. Nota di copertura: nessun test a livello repository esercita la SQL reale di delete per nessuna delle tre entità (i test di service mockano `repository.delete()`) — il fix è verificato per compilazione e build (`./gradlew :frontdesk-service:test` verde) ma non da un test che dimostri esplicitamente il `409` su una delete stale, da pianificare in un giro successivo.

Chiuso 2026-08-16: nuovo `SoftDeleteVersionCheckIntegrationTest` — Testcontainers PostgreSQL reale, stesso setup di `RoomTypeServiceIntegrationTest`, nessun mock su `repository.delete()`. Per ciascuna delle tre entità: carica la riga, incrementa `version` nel database con una `UPDATE` SQL nativa (simula un salvataggio concorrente che l'istanza in memoria non vede), poi tenta la delete con la `version` ormai stale — verifica che sollevi `ObjectOptimisticLockingFailureException`. Verificato che il test cattura davvero la regressione, non solo che compila: rimosso temporaneamente `check = ResultCheckStyle.COUNT` dalle tre entità e rieseguito — tutti e 3 i test falliscono sull'asserzione (nessuna eccezione sollevata, la delete stale "riesce" silenziosamente, esattamente il sintomo descritto sopra); ripristinato il fix e rieseguito — tutti e 3 tornano verdi, insieme all'intera build (`checkstyle/PMD/SpotBugs/jacoco`). Commit: `fix(security): version-check soft-deletes on Quotation/RateSeason, enforce it on all three (GAP-18)`, `test(security): repository-level stale-delete race test for GAP-18`

A.3 Configurazione e SDLC avanzati

T-CFG-03 — Autenticazione HTTP Basic sul config-service Il config-service (porta 8888) espone JWT_SECRET, POSTGRES_PASSWORD e INTERNAL_HMAC_SECRET. HTTP Basic Auth aggiunta per impedire lettura anonima. Commit: `fix(security): T-CFG-03 -- HTTP Basic Auth on config-service`

SAST/CodeQL e supply chain CI/CD (OWASP SAMM Verification) 10 alert CodeQL risolti: permessi minimi `contents: read` su ogni job CI (OWASP A02:2025 — Security Misconfiguration), 9 falsi positivi CSRF su microservizi interni chiusi con motivazione. Corrisponde alla fase **Verification** di OWASP SAMM e al principio shift-left. Commit: `fix(ci): add permissions: contents: read to all CI jobs`

3 ulteriori alert `java/log-injection` (2026-07-31) su `internal-auth-lib/InternalAuthFilter.ja` — falsi positivi: `sanitizeForLog()` rimuove già CR/LF (CWE-117) prima di ogni chiamata di log flaggata, ma la query standard di CodeQL non riconosce un sanitizzatore custom del progetto come interruzione del flusso di taint. Commenti di soppressione `codeql[java/log-injection]` aggiunti ad ogni chiamata (documentazione inline, non solo dismissal silenzioso); alert chiusi su GitHub con motivazione.

CVE scanning con Trivy (DEP-CVE-01/02/04) Pipeline CI esegue Trivy su ogni immagine Docker. CVE risolte: `CVE-2026-33870/33871` (Netty request smuggling + HTTP/2 DoS), `CVE-2025-41235/41253` (Spring Cloud Gateway header forwarding + EL injection). 108 alert su immagini terze parti triaggiati con motivazione documentata.

Aggiornamento dipendenze frontend (DEP-CVE-05) L'analisi del security tab GitHub ha rilevato 7 CVE sulle dependenze npm del frontend. Risolte con upgrade mirati senza breaking change (OWASP A06:2025):

- `react-router-dom` + `react-router` `7.13.1` → `7.17.0`: `#36` XSS via RSC redirect, `#37` XSS via `Location` header, `#38` RCE via turbo-stream, `#39` open redirect, `#40`/`#41` DoS via `__manifest` e single-fetch.
- `esbuild` `0.27.3` → `0.28.1` (transitivo di Vite, forzato via `npm overrides`): `CVE-2025-42577` arbitrary file read tramite dev server su Windows.

Tutti i 344 test rimasti verdi; build e lint zero warning. Commit: `fix(deps): resolve 7 Dependabot CVEs -- react-router 7.17.0, esbuild 0.28.1`

Appendice B

Appendice B — Utilizzo dell'IA e Revisione Critica

B.1 Metodologia

Il progetto è stato sviluppato a partire da una codebase personale preesistente (branch `pre-secure-coding`), usata come baseline insicura. **Claude Code** (Anthropic, modello `claude-sonnet-4-6`) è stato impiegato come strumento di analisi e supporto all'implementazione, *non* come sostituto del ragionamento critico dello studente.

In questa appendice sono documentati i momenti in cui l'output dell'AI è stato valutato criticamente, evidenziando i casi in cui il suggerimento era errato, incompleto o incompatibile con il contesto specifico del sistema.

B.2 Episodio 1 — JWT storage: localStorage suggerito dall'AI

Prompt utilizzato

> "Implementa un sistema di autenticazione JWT con Spring Boot e React. Fornisci il codice per generare il token nel backend e per archiviarlo nel frontend."

Output dell'AI L'AI ha generato un'implementazione funzionante dove il frontend archiviava il token con `localStorage.setItem('token', response.data.token)` e lo inviava come header `Authorization: Bearer <token>` su ogni richiesta Axios.

Critica allo studente `localStorage` è accessibile a qualsiasi script JavaScript della pagina. Un attacco XSS — anche tramite una dipendenza npm compromessa — può leggere il token con `localStorage.getItem('token')` e inviarlo a un server dell'attaccante. L'AI ha prodotto codice funzionale ma insicuro per OWASP A07 (Identification and Authentication Failures). Il problema non è nell'algoritmo JWT ma nel vettore di esfiltrazione abilitato dallo storage scelto.

Correzione applicata (T-FE-02 / T-AUTH-04) Il token è stato spostato in un cookie `httpOnly` impostato dal server:

Cookie httpOnly al posto di localStorage (T-FE-02)

```
// Il browser non puo' leggere il cookie con JavaScript (difesa XSS)
private ResponseCookie createCookie(String value, int maxAge) {
    return ResponseCookie.from("jwt", value)
        .httpOnly(true)    // blocca lettura da JavaScript
        .secure(true)      // trasmesso solo su HTTPS
        .sameSite("Strict") // blocca invio cross-site
        .path("/")
        .maxAge(maxAge)
        .build();
}
```

B.3 Episodio 2 — BCrypt: cost factor obsoleto per il 2025/26

Prompt utilizzato

> "Sostituisci la memorizzazione della password in chiaro con un hashing sicuro. Usa BCrypt con Spring Security."

Output dell'AI L'AI ha generato `new BCryptPasswordEncoder()` senza parametri, che usa il cost factor di default di Spring Security: **10**. Il codice era strutturalmente corretto ma usava il parametro di default.

Critica allo studente Il cost factor 10 era adeguato per hardware del 2011. Le raccomandazioni OWASP al 2025/26 indicano cost ≥ 12 come minimo su hardware moderno, per garantire almeno 300 ms di calcolo su singolo core. Con cost 10 su hardware attuale il calcolo richiede ~50 ms, rendendo il brute force $2^2 = 4\times$ più rapido del necessario (SEI CERT MSC62-J).

Correzione applicata (T-AUTH-03)

BCrypt cost 12 esplicito (T-AUTH-03, SEI CERT MSC62-J)

```
// Cost 12: OWASP raccomanda >= 12 per hardware 2025.
// Cost 10 (default Spring Security) era ottimale per hardware 2011.
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(12);
}
```

B.4 Episodio 3 — IDOR: l'AI ha ignorato il contesto multi-tenant

Prompt utilizzato

> "Nel guest-service, un utente può accedere ai dati di ospiti che non appartengono alla sua struttura. Correggi il problema IDOR."

Output dell'AI L'AI ha suggerito di aggiungere un controllo basato sull'`userId` del chiamante, verificando `guest.createdBy == currentUserId`:

```
// Suggerimento AI -- ERRATO nel contesto multi-hotel
Optional<Guest> findByIdAndCreatedBy(UUID guestId, UUID userId);
```

Critica allo studente Il suggerimento era errato nel contesto di un PMS multi-hotel. Le risorse appartengono a un *hotel*, non a un singolo utente. Con il modello proposto dall'AI: un ADMIN non potrebbe accedere agli ospiti creati da un RECEPTIONIST dello stesso hotel; un RECEPTIONIST trasferito manterrebbe accesso agli ospiti del precedente. La dimensione di isolamento corretta è `hotel_id` estratto dal JWT, non `userId`.

Correzione applicata (T-GST-01 / T-GST-03)

Isolamento per hotel_id (T-GST-01, OWASP A01)

```
// Isolamento per hotel, non per utente
Optional<Guest> findByIdAndActiveTrueAndHotelId(UUID id, UUID hotelId);

// hotelId estratto dal SecurityContext, mai dal client (IDOR-safe)
private UUID resolveHotelId() {
    Authentication auth =
        SecurityContextHolder.getContext().getAuthentication();
    return UUID.fromString((String) auth.getDetails());
}
```

B.5 Episodio 4 — CSRF: pattern incompatibile con SPA React

Prompt utilizzato

> "Implementa la protezione CSRF per il gateway API. L'applicazione usa cookie httpOnly per il JWT e Spring Cloud Gateway (WebFlux)."

Output dell'AI L'AI ha suggerito il filtro CSRF standard di Spring Security con `CookieCsrfTokenRepository.withHttpOnlyFalse()`, che attende il token in un campo `_csrf` del form HTML o nell'header `X-CSRF-TOKEN`.

Critica allo studente Il filtro standard Spring Security è progettato per applicazioni MVC che rendono form HTML. Non è compatibile con una SPA React che usa chiamate Axios JSON perché: (1) il gateway è Spring Cloud Gateway (WebFlux/reactive), non Spring MVC; (2) il token CSRF viene esposto solo su endpoint specifici con richiesta asincrona aggiuntiva, incompatibile con l'architettura SPA senza rendering server-side.

Correzione applicata (T-GW-05) Implementato il pattern **Double Submit Cookie** con un filtro personalizzato: il cookie `csrf_token` (non-httpOnly, leggibile dalla SPA) viene confrontato con l'header `X-CSRF-Token` (aggiunto da Axios). Un sito malevolo non può costruire l'header perché la same-origin policy impedisce di leggere il cookie.

B.6 Episodio 5 — User enumeration: messaggi "utili per il debug"

Prompt utilizzato

> "Scrivi il metodo login() per AuthServiceImpl. Distingui i casi errore per facilitare il debug: utente non trovato e password errata."

Output dell'AI

```
// Output AI -- INSICURO: messaggi distinti
.orElseThrow(() ->
    new BadCredentialsException("USER_NOT_FOUND")); // diverso!

if (!passwordEncoder.matches(...)) {
    throw new BadCredentialsException("INVALID_PASSWORD"); // diverso!
}
```

Critica allo studente Messaggi distinti consentono la *user enumeration*: un attaccante può inviare richieste automatizzate e identificare username validi dalla risposta. OWASP A07 cita esplicitamente questo pattern come vettore per brute force mirati. **L'AI ha motivato la scelta come "utile per il debug"**, dimostrando che non considera il contesto di sicurezza nella formulazione dei messaggi di errore.

Correzione applicata (T-AUTH-01)

Messaggio uniforme — nessuna information disclosure (T-AUTH-01)

```
// SICURO: identico per USER_NOT_FOUND e INVALID_PASSWORD
.orElseThrow(() ->
    new BadCredentialsException("INVALID_CREDENTIALS"));

if (!passwordEncoder.matches(request.password(),
                             user.getPasswordHash())) {
    throw new BadCredentialsException("INVALID_CREDENTIALS");
}
```

B.7 Riepilogo della revisione critica

Episodio	Errore/limite dell'AI	Correzione studente
JWT storage	`localStorage` (vulnerabile a XSS)	Cookie `httpOnly` + `SameSite=Strict`
BCrypt cost factor	Default 10 (hardware 2011)	Cost 12 esplicito (OWASP 2025)
IDOR multi-tenant	Isolamento per `userId` (errato)	Isolamento per `hotel_id` (corretto)
CSRF su SPA	Filter Spring MVC (incompatibile WebFlux)	Double Submit Cookie custom
User enumeration	Messaggi distinti "per debug"	Messaggio uniforme `INVALID_CREDENTIALS`

In tutti e cinque i casi l'AI ha prodotto codice *funzionale* ma con difetti di sicurezza identificabili applicando i principi del corso: OWASP Top 10, SEI CERT e la conoscenza del contesto architetturale del sistema.