SOMMARIO

[1) Introduzione al Sistema](#_Toc9747311)

[1.1 Scopo e Contesto del Progetto](#_Toc9747312)

[1.2 Panoramica dei Componenti Principali](#_Toc9747313)

[1.3 Dipendenze e Tecnologie Utilizzate](#_Toc9747314)

[1.4 Interazione e Flusso dei Componenti](#_Toc9747315)

[2) Descrizione Generale del Sistema](#_Toc9747316)

[2.1 Logica e Struttura del Sistema](#_Toc9747317)

[2.2 Interazioni e Flussi](#_Toc9747318)

[2.3 Comportamenti Condivisi e Design Pattern](#_Toc9747319)

[2.4 Scalabilità e Modularità (Avanzato)](#_Toc97473110)

[2.5 Suggerimento Diagramma (Opzionale)](#_Toc97473111)

[3) Struttura delle Classi](#_Toc97473112)

[3.1 Pattern di Attributi e Dipendenze](#_Toc97473113)

[3.2 Scelte di Design e Annotazioni](#_Toc97473114)

[3.3 Ruoli e Responsabilità per Layer](#_Toc97473115)

[3.4 Relazioni tra Classi](#_Toc97473116)

[3.5 Suggerimento per Diagramma UML delle Classi](#_Toc97473117)

[4) Panoramica dei Metodi](#_Toc97473118)

[4.1 Raggruppamento per Funzionalità](#_Toc97473119)

[4.2 Gestione delle Eccezioni](#_Toc97473120)

[4.3 Flusso Tipico di Esecuzione](#_Toc97473121)

[4.4 Casi d'Uso Realistici](#_Toc97473122)

[5) Usage Examples](#_Toc97473123)

[5.1 Identificazione dei Contesti Comuni di Utilizzo](#_Toc97473124)

[5.2 Esempi Pratici e Concisi](#_Toc97473125)

[5.3 Flusso Integrato Tipico](#_Toc97473126)

[5.4 Contesto Prerequisito](#_Toc97473127)

[6) Exception Handling](#_Toc97473128)

[6.1 Mappa delle Eccezioni Custom](#_Toc97473129)

[6.2 Strategia Globale di Gestione Errori](#_Toc97473130)

[6.3 Logica di Recovery e Fallback](#_Toc97473131)

[6.4 Raccomandazioni per gli Sviluppatori](#_Toc97473132)

[7) Best Practices & Improvements](#_Toc97473133)

[7.1 Code Quality & Design](#_Toc97473134)

[7.2 Performance Optimizations](#_Toc97473135)

[7.3 Security Improvements](#_Toc97473136)

[7.4 Development Practices](#_Toc97473137)

[7.5 Refactoring Opportunities](#_Toc97473138)

[8) Struttura del Database](#_Toc97473139)

[8.1 Modellazione delle Entità e Relazioni](#_Toc97473140)

[8.2 Progettazione dei Repository e Pattern di Accesso](#_Toc97473141)

[8.3 Gestione delle Transazioni](#_Toc97473142)

[8.4 Suggerimento per Diagramma dello Schema del Database](#_Toc97473143)

[9) Descrizione delle API](#_Toc97473144)

[9.1 Panoramica degli Endpoint](#_Toc97473145)

[9.2 Modelli di Richiesta e Risposta](#_Toc97473146)

[9.3 Sicurezza e Autorizzazione](#_Toc97473147)

[9.4 Suggerimento per Diagramma di Flusso API](#_Toc97473148)

## 1) Introduzione al Sistema

### 1.1 Scopo e Contesto del Progetto

Il sistema è progettato per gestire un modulo anagrafica completo, focalizzato sulla gestione di entità "Persona" e delle relative informazioni correlate quali indirizzi, documenti e luoghi di nascita. L’obiettivo principale è fornire un backend robusto e scalabile, basato su Spring Boot, che consenta operazioni CRUD, ricerche avanzate e aggiornamenti coerenti delle informazioni anagrafiche. Il sistema si rivolge a contesti in cui è necessario mantenere dati anagrafici accurati e integrati, supportando scenari tipici come la registrazione, modifica e consultazione di dati personali, indirizzi di residenza e domicilio, documenti identificativi e dati geografici associati.

### 1.2 Panoramica dei Componenti Principali

L’architettura si articola in diversi livelli e componenti chiave:

* **Controller REST**: PersonaController, IndirizzoController, DocumentoController, LuogoController espongono API REST per l’interazione con il sistema, gestendo richieste HTTP e delegando la logica di business ai servizi. Sono configurati con prefissi dinamici e integrano validazione e documentazione OpenAPI.
* **Service Layer**: Interfacce e implementazioni come PersonaService e PersonaServiceImpl, IndirizzoService, DocumentoService, LuogoService orchestrano la logica di business, gestendo operazioni CRUD, validazioni di dominio e coordinamento tra repository e mapper.
* **Repository**: Interfacce Spring Data JPA (PersonaRepository, IndirizzoRepository, LuogoRepository) forniscono l’accesso ai dati persistenti, ereditando metodi CRUD e supportando query personalizzate, garantendo l’astrazione del livello di persistenza.
* **Mapper**: Componenti basati su MapStruct (PersonaMapper, IndirizzoMapper, DocumentoMapper, LuogoMapper) gestiscono la conversione bidirezionale tra entità JPA e DTO, favorendo la separazione tra modello di dominio e rappresentazione dati trasferita tra livelli.
* **Entità di Dominio**: Classi JPA (Persona, Indirizzo, Documento, Luogo) modellano le strutture dati persistenti, con relazioni ben definite e annotazioni per la mappatura su database relazionale.
* **DTO (Data Transfer Object)**: Classi come PersonaDtoInput, PersonaDtoOutput, IndirizzoDto, DocumentoDto, LuogoDto incapsulano i dati scambiati tra client e server, garantendo sicurezza e decoupling dal modello di dominio.
* **Gestione delle Eccezioni**: Classi personalizzate (BusinessException, DuplicateEntityException, NotFoundException) e componenti di supporto (ErrorCode, ErrorResponse, GlobalExceptionHandler) centralizzano la gestione degli errori, standardizzando risposte e codici di stato HTTP.
* **Configurazione**: CorsConfig gestisce le politiche CORS per la sicurezza e l’accessibilità delle API da origini esterne, mentre i file di configurazione application.yml e application.properties definiscono parametri di sistema, datasource, logging e proprietà specifiche.
* **Avvio Applicazione**: AnagraficaApplication funge da punto di ingresso, avviando il contesto Spring Boot e abilitando component scanning e configurazioni automatiche.

### 1.3 Dipendenze e Tecnologie Utilizzate

Il progetto si basa su un ecosistema tecnologico consolidato per applicazioni enterprise Java:

* **Spring Boot**: Framework principale per la configurazione, gestione del ciclo di vita dei bean, sicurezza, web MVC e integrazione con JPA.
* **Spring Data JPA**: Per la persistenza dei dati, semplificando l’accesso al database relazionale tramite repository e query automatiche.
* **Hibernate**: Provider JPA utilizzato per il mapping ORM tra entità Java e tabelle del database.
* **MapStruct**: Framework di mapping automatico per la conversione tra entità e DTO, integrato con Spring per la gestione dei bean.
* **Lombok**: Libreria per la generazione automatica di codice boilerplate (getter, setter, costruttori, metodi equals, hashCode, toString), migliorando la leggibilità e la manutenzione del codice.
* **Jakarta Validation (JSR-380)**: Per la validazione dei dati in ingresso nei DTO, garantendo integrità e coerenza prima della logica di business.
* **Swagger/OpenAPI**: Per la documentazione automatica delle API REST, facilitando l’interazione e la comprensione da parte di sviluppatori e client esterni.
* **H2 Database**: Database in-memory utilizzato per sviluppo e test, configurato tramite application.yml.
* **Maven**: Sistema di build e gestione delle dipendenze, configurato tramite pom.xml con versioni specifiche di Java (21) e Spring Boot (3.5.3).

### 1.4 Interazione e Flusso dei Componenti

Il flusso tipico di una richiesta nel sistema inizia con una chiamata HTTP indirizzata a uno dei controller REST (PersonaController, IndirizzoController, ecc.), che riceve i dati in ingresso sotto forma di DTO validati. Il controller delega quindi al corrispondente servizio di business (PersonaServiceImpl, IndirizzoService, ecc.) che applica la logica di dominio, interagendo con i repository per accedere o modificare i dati persistenti. Durante questo processo, i mapper MapStruct convertono i dati tra entità JPA e DTO per mantenere la separazione dei livelli. In caso di errori o condizioni particolari, le eccezioni personalizzate vengono lanciate e gestite centralmente da GlobalExceptionHandler, che traduce le eccezioni in risposte HTTP strutturate con codici e messaggi coerenti. La configurazione CORS garantisce che le API siano accessibili in sicurezza da origini autorizzate.

Note: Per rappresentare visivamente questo flusso, un diagramma di componenti evidenziando i layer Controller, Service, Repository, Mapper e Entity sarebbe ideale. Un diagramma di sequenza potrebbe inoltre illustrare la chiamata HTTP, la validazione, la conversione DTO-Entity, l’accesso al database e la gestione delle eccezioni in un ciclo completo di richiesta-risposta.

## 2) Descrizione Generale del Sistema

### 2.1 Logica e Struttura del Sistema

Il sistema è progettato secondo un'architettura a strati tipica delle applicazioni Spring Boot, che separa chiaramente i livelli di presentazione, logica di business e persistenza. Al livello più alto, i controller REST espongono API per la gestione delle entità anagrafiche, documenti, indirizzi e luoghi di nascita, fungendo da interfaccia verso i client esterni. Sotto i controller, il Service Layer incapsula la logica di business, orchestrando le operazioni e garantendo la coerenza dei dati. Questo livello utilizza pattern consolidati come il Service Layer e l’Injection delle dipendenze per mantenere il codice modulare e testabile. Il livello di persistenza è gestito tramite repository JPA, che astraggono l’accesso al database relazionale e supportano operazioni CRUD e query avanzate. I modelli dati sono rappresentati da entità JPA, mentre i DTO (Data Transfer Object) facilitano la separazione tra modello di dominio e rappresentazione dati per la comunicazione esterna.

### 2.2 Interazioni e Flussi

Le componenti comunicano principalmente in modo sincrono tramite chiamate REST HTTP, con i controller che ricevono le richieste e le delegano ai servizi corrispondenti. I servizi interagiscono con i repository per il recupero e la persistenza dei dati, e con i mapper per la conversione tra entità e DTO, garantendo così una netta separazione tra livelli. I flussi principali includono operazioni CRUD su persone, documenti, indirizzi e luoghi di nascita, con validazioni e regole di business applicate nel Service Layer. La configurazione CORS consente richieste cross-origin da origini specifiche, assicurando la sicurezza e la compatibilità con client web esterni. L’integrazione con il database H2 in memoria facilita lo sviluppo e i test, mentre la gestione centralizzata delle eccezioni garantisce risposte coerenti e strutturate verso i client.

### 2.3 Comportamenti Condivisi e Design Pattern

Il sistema adotta pattern architetturali consolidati come il Service Layer per isolare la logica di business, il Repository per l’accesso ai dati e il DTO per la separazione tra modello di dominio e rappresentazione dati. I mapper, implementati tramite MapStruct, automatizzano la conversione tra entità e DTO, migliorando la manutenibilità e riducendo il codice boilerplate. L’iniezione delle dipendenze, facilitata da Lombok e Spring, promuove un codice pulito e facilmente testabile. La gestione centralizzata delle eccezioni tramite un handler globale consente di uniformare la risposta agli errori, utilizzando eccezioni specifiche come BusinessException, NotFoundException e DuplicateEntityException per migliorare la chiarezza e la robustezza del sistema. La configurazione CORS è implementata come bean singleton, applicata globalmente per garantire la sicurezza delle richieste HTTP.

### 2.4 Scalabilità e Modularità (Avanzato)

L’architettura modulare, basata su componenti chiaramente separati e interfacce ben definite, facilita l’estensione e la sostituzione di singoli moduli senza impatti trasversali. L’uso di repository JPA e DTO consente di adattare facilmente il modello dati o la rappresentazione esterna senza modifiche invasive. La configurazione del database in memoria H2 è ideale per ambienti di sviluppo e test, mentre l’adozione di Spring Boot permette un facile passaggio a database più robusti in produzione. La gestione centralizzata degli errori e la configurazione CORS contribuiscono alla resilienza e alla sicurezza del sistema. Sebbene non siano presenti meccanismi espliciti di caching o bilanciamento del carico, l’architettura a strati e la separazione delle responsabilità supportano l’adozione futura di strategie di scalabilità orizzontale o verticale.

### 2.5 Suggerimento Diagramma (Opzionale)

Nota: Si consiglia di realizzare un diagramma a blocchi che evidenzi i principali componenti architetturali: i controller REST, il Service Layer, i repository JPA e i mapper. Il diagramma dovrebbe mostrare i flussi di dati tra client esterni e controller, la delega della logica di business ai servizi, l’interazione con il livello di persistenza e la conversione dati tramite mapper. Un diagramma di sequenza potrebbe inoltre illustrare il flusso tipico di una richiesta CRUD, evidenziando la separazione tra livelli e la gestione centralizzata delle eccezioni.

## 3) Struttura delle Classi

### 3.1 Pattern di Attributi e Dipendenze

Le classi del sistema mostrano una chiara separazione tra dati, logica di business e persistenza, con un uso consistente di repository, mapper e servizi. Gli attributi ricorrenti includono repository JPA (PersonaRepository, IndirizzoRepository, LuogoRepository) per l’accesso ai dati, e mapper (es. PersonaMapper, DocumentoMapper, IndirizzoMapper, LuogoMapper) per la conversione tra entità e DTO.

Le dipendenze sono prevalentemente iniettate tramite annotazioni Spring @Autowired su campi, sebbene in alcuni casi sia presente anche l’uso di Lombok @RequiredArgsConstructor per constructor injection, anche se non sempre applicata coerentemente.

I DTO sono strutture dati semplici, prive di logica, con attributi annotati da Lombok (@Data, @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor) per ridurre il boilerplate. Le entità JPA sono dotate di annotazioni per la mappatura su tabelle e relazioni, con campi tipici per rappresentare persone, indirizzi, documenti e luoghi.

### 3.2 Scelte di Design e Annotazioni

Il sistema segue principi di separazione delle responsabilità e design orientato alle interfacce. Le interfacce di servizio (PersonaService) definiscono i contratti senza implementazione, favorendo l’astrazione e la testabilità.

Le classi di servizio sono annotate con @Service e gestiscono la logica di business, delegando a repository e mapper. I controller REST sono contrassegnati da @RestController e utilizzano annotazioni di mapping HTTP (@GetMapping, @PostMapping, ecc.) e validazione (@Valid), integrando anche annotazioni per la documentazione OpenAPI (@Operation, @ApiResponses).

I mapper sono interfacce annotate con MapStruct (@Mapper(componentModel = "spring")), che generano automaticamente implementazioni per la conversione tra DTO e entity, mantenendo il codice pulito e modulare.

Le eccezioni custom estendono RuntimeException e sono semplici, mentre la gestione centralizzata degli errori è affidata a classi annotate con @RestControllerAdvice che intercettano eccezioni specifiche e generiche, restituendo risposte standardizzate.

### 3.3 Ruoli e Responsabilità per Layer

* **Controller (Presentation Layer):** espongono API REST, gestiscono richieste HTTP, validano input e orchestrano chiamate ai servizi. Sono organizzati per risorse (es. PersonaController, DocumentoController, IndirizzoController, LuogoController) e utilizzano DTO per input/output.
* **Service (Business Logic Layer):** implementano la logica applicativa, coordinano operazioni tra repository e mapper, e garantiscono la coerenza dei dati. Servizi come PersonaServiceImpl, DocumentoService, IndirizzoService e LuogoService gestiscono rispettivamente persone, documenti, indirizzi e luoghi.
* **Repository (Data Access Layer):** interfacce che estendono JpaRepository per fornire metodi CRUD e query personalizzate, senza implementazioni manuali. Gestiscono la persistenza delle entità Persona, Indirizzo, Luogo, Documento.
* **DTO e Mapper:** i DTO rappresentano dati di input e output senza logica, mentre i mapper (MapStruct) traducono tra DTO e entità, mantenendo separati i modelli di dominio e di trasporto dati.
* **Configurazione:** classi come quella per la configurazione CORS (@Configuration) gestiscono aspetti trasversali dell’applicazione.
* **Gestione Eccezioni:** classi con @RestControllerAdvice centralizzano il trattamento degli errori, migliorando la robustezza e la coerenza delle risposte.

### 3.4 Relazioni tra Classi

Le entità JPA sono collegate tramite relazioni @ManyToOne (es. Persona associa Luogo, Indirizzo, Documento), modellando un dominio normalizzato.

I servizi implementano interfacce di servizio, garantendo un design orientato ai contratti. I controller dipendono dai servizi tramite composizione.

I mapper sono interfacce senza implementazioni manuali, generate da MapStruct, e collaborano con DTO ed entità.

I repository estendono JpaRepository e sono utilizzati dai servizi per l’accesso ai dati.

Le classi di eccezione estendono RuntimeException e sono utilizzate dal layer di gestione errori.

Non sono presenti ereditarietà complesse o gerarchie profonde, ma una struttura a componenti ben definita e modulare.

### 3.5 Suggerimento per Diagramma UML delle Classi

Note: Un diagramma UML testuale potrebbe rappresentare i principali componenti come segue:

* PersonaController dipende da PersonaService
* PersonaServiceImpl implementa PersonaService e dipende da PersonaRepository e PersonaMapper
* PersonaRepository estende JpaRepository<Persona, Long>
* Persona entità JPA con relazioni ManyToOne verso Luogo, Indirizzo, Documento
* Mapper come PersonaMapper, DocumentoMapper, IndirizzoMapper, LuogoMapper sono interfacce con metodi di conversione DTO-entity
* Servizi DocumentoService, IndirizzoService, LuogoService dipendono rispettivamente da repository e mapper specifici
* Controller REST per ciascuna risorsa (es. DocumentoController, IndirizzoController) dipendono dai relativi servizi
* Classe di configurazione CORS è un componente isolato annotato con @Configuration
* Gestione eccezioni centralizzata tramite classe annotata @RestControllerAdvice che intercetta eccezioni custom (BusinessException, NotFoundException, DuplicateEntityException)

Questa struttura evidenzia un’architettura stratificata, con chiara separazione tra livelli di presentazione, logica di business, persistenza e mapping dati.

## 4) Panoramica dei Metodi

### 4.1 Raggruppamento per Funzionalità

**Ricerca e Recupero Dati**

* **Tipica firma:**  
  List<PersonaDtoOutput> cercaPersone(Optional<String> nome, Optional<String> cognome, Optional<String> codiceFiscale, Optional<String> comuneNascita, Optional<String> provinciaNascita, Optional<String> statoNascita)  
  PersonaDtoOutput getPersonaById(Long id)
* **Scopo:** Permettono di cercare persone tramite filtri opzionali o recuperare una persona tramite ID univoco.
* **Parametri:** Filtri di tipo Optional<String> per ricerca; Long per ID.
* **Return:** Liste di DTO o singolo DTO di persona.
* **Comportamento:** Gestione flessibile dei filtri, restituzione di dati mappati in DTO per separazione tra dominio e API.

**Creazione, Aggiornamento ed Eliminazione (CRUD)**

* **Tipica firma:**  
  PersonaDtoOutput creaPersona(PersonaDtoInput personaDtoInput)  
  PersonaDtoOutput aggiornaPersona(Long id, PersonaDtoInput personaDtoInput)  
  void eliminaPersona(Long id)
* **Scopo:** Gestione completa del ciclo di vita di una persona: creazione, modifica e cancellazione.
* **Parametri:** DTO di input per creazione/aggiornamento, ID per individuare la risorsa.
* **Return:** DTO aggiornati o void per eliminazione.
* **Comportamento:** Validazione dati, persistenza, e gestione dell’esistenza della risorsa.

**Aggiornamento di Dati Collegati (Indirizzi, Documenti, Luoghi)**

* **Tipica firma:**  
  IndirizzoDto aggiornaResidenza(Long personaId, IndirizzoDto indirizzoDto)  
  IndirizzoDto aggiornaDomicilio(Long personaId, IndirizzoDto indirizzoDto)  
  DocumentoDto aggiornaDocumento(Long idPersona, DocumentoDto dto)  
  LuogoDto aggiornaLuogoNascita(Long personaId, LuogoDto luogoDto)
* **Scopo:** Modificare informazioni collegate alla persona quali indirizzo di residenza, domicilio, documenti e luogo di nascita.
* **Parametri:** ID persona e DTO specifici per il dato da aggiornare.
* **Return:** DTO aggiornati corrispondenti all’entità modificata.
* **Comportamento:** Recupero entità, mapping DTO-entità, salvataggio e gestione errori se persona non trovata.

**Configurazione e Supporto Tecnico**

* **Tipica firma:**  
  CorsFilter corsFilter()
* **Scopo:** Configurare aspetti tecnici come il filtro CORS per abilitare richieste cross-origin.
* **Parametri:** Nessuno.
* **Return:** Istanza di filtro configurata.
* **Comportamento:** Definizione di regole di sicurezza e accesso per le API.

### 4.2 Gestione delle Eccezioni

* **Eccezioni comuni:**
  + BusinessException con codici specifici (es. PERSONA\_NOT\_FOUND) per indicare errori di business come entità non trovate o dati non validi.
  + EntityNotFoundException o eccezioni simili per assenza di risorse nel database.
  + DuplicateEntityException per tentativi di inserimento di dati duplicati (es. codice fiscale già presente).
  + Validazioni automatiche tramite annotazioni @Valid possono generare errori 400 in caso di dati non conformi.
  + Errori generici o inattesi sono gestiti da handler globali che restituiscono risposte di errore standardizzate.
* **Condizioni di trigger:**
  + Ricerca o aggiornamento di persona inesistente.
  + Inserimento di dati duplicati o non validi.
  + Problemi di persistenza o mapping dati.
  + Richieste non autorizzate o malformate (es. CORS non configurato correttamente).

### 4.3 Flusso Tipico di Esecuzione

Note: Un flusso comune per la gestione di una persona prevede la sequenza seguente:

1. **Chiamata Controller**: riceve la richiesta REST con parametri o payload DTO.
2. **Validazione Input**: annotazioni @Valid e controlli custom verificano la correttezza dei dati.
3. **Invocazione Service**: il service esegue la logica di business, recupera entità dal repository o crea nuove entità.
4. **Mapping DTO-Entità**: conversione tra DTO e entità tramite mapper (es. MapStruct).
5. **Persistenza**: salvataggio o aggiornamento tramite repository JPA.
6. **Gestione Eccezioni**: eventuali errori sono intercettati e trasformati in risposte HTTP appropriate.
7. **Risposta**: ritorno di DTO di output incapsulati in ResponseEntity con status HTTP.

**Pseudocodice semplificato:**

// Esempio aggiornamento persona
public ResponseEntity<PersonaDtoOutput> aggiornaPersona(Long id, PersonaDtoInput input) {
Persona persona = repository.findById(id)
.orElseThrow(() -> new BusinessException(PERSONA\_NOT\_FOUND));
mapper.updateEntityFromDto(input, persona);
Persona saved = repository.save(persona);
PersonaDtoOutput dtoOut = mapper.toDtoOut(saved);
return ResponseEntity.ok(dtoOut);
}

### 4.4 Casi d'Uso Realistici

**Ricerca persone con filtro nome:**

List<PersonaDtoOutput> risultati = personaService.cercaPersone(
Optional.of("Mario"),
Optional.empty(),
Optional.empty(),
Optional.empty(),
Optional.empty(),
Optional.empty()
);

**Recupero persona per ID:**

PersonaDtoOutput persona = personaService.getPersonaById(123L);
System.out.println("Nome: " + persona.getNome());

**Creazione nuova persona:**

PersonaDtoInput input = new PersonaDtoInput(
"Mario", "Rossi", "M", "RSSMRA80A01H501U", documentoDto,
LocalDate.of(1980, 1, 1), "Italiana", indirizzoDto, luogoDto,
"Coniugato", "Laurea", "Ingegnere", 2,
"mario.rossi@example.com", "1234567890", "Nessuna nota"
);
PersonaDtoOutput nuovaPersona = personaService.creaPersona(input);

**Aggiornamento indirizzo di residenza:**

IndirizzoDto nuovoIndirizzo = new IndirizzoDto("Via Roma", "10", "00100", "Roma", "RM", "Italia");
IndirizzoDto residenzaAggiornata = indirizzoService.aggiornaResidenza(123L, nuovoIndirizzo);

**Aggiornamento documento associato a persona:**

DocumentoDto documentoDto = new DocumentoDto("Carta d'Identità", "AB1234567", "Comune di Milano",
LocalDate.of(2020, 5, 20), LocalDate.of(2030, 5, 20));
DocumentoDto documentoAggiornato = documentoService.aggiornaDocumento(123L, documentoDto);

**Configurazione filtro CORS per frontend Angular:**

@Bean
public CorsFilter corsFilter() {
CorsConfiguration config = new CorsConfiguration();
config.addAllowedOrigin("http://localhost:4200");
config.addAllowedMethod("GET");
config.addAllowedMethod("POST");
config.addAllowedMethod("PUT");
config.addAllowedMethod("PATCH");
config.addAllowedMethod("DELETE");
config.addAllowedMethod("OPTIONS");
config.addAllowedHeader("\*");
config.setAllowCredentials(true);
UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/\*\*", config);
return new CorsFilter(source);
}

Questa panoramica sintetizza i metodi pubblici chiave del sistema, raggruppandoli per scopo funzionale e illustrandone l’uso pratico, la gestione delle eccezioni e il flusso tipico di esecuzione, facilitando così la comprensione e l’adozione da parte degli sviluppatori nel contesto ATE.

## 5) Usage Examples

### 5.1 Identificazione dei Contesti Comuni di Utilizzo

Gli esempi raccolti mostrano l’uso tipico dei componenti del sistema raggruppati per livello applicativo o caso d’uso:

* **Controller Layer**: gestione delle richieste REST, mapping degli endpoint, validazione input, e delega ai servizi di business.
* **Service Layer**: implementazione della logica di business, gestione delle transazioni, validazioni, mapping tra DTO ed entità, e interazione con i repository.
* **Repository Layer**: accesso ai dati tramite Spring Data JPA, con metodi CRUD e query personalizzate.
* **DTO e Mapper**: trasferimento dati tra livelli, conversione tra entità e DTO per isolare la persistenza dalla rappresentazione esterna.
* **Gestione errori**: uso di eccezioni custom (BusinessException, DuplicateEntityException) e handler globali (@ControllerAdvice) per risposte REST uniformi.
* **Configurazioni di supporto**: gestione CORS, caricamento file, configurazioni di database in memoria (H2), e avvio dell’applicazione Spring Boot.

### 5.2 Esempi Pratici e Concisi

#### Controller REST per la gestione di Persona

@RestController
@RequestMapping("/api/persone")
public class PersonaController {
private final PersonaService personaService;
public PersonaController(PersonaService personaService) {
this.personaService = personaService;
}
@GetMapping
public List<PersonaDtoOutput> cercaPersone(
@RequestParam Optional<String> nome,
@RequestParam Optional<String> cognome,
@RequestParam Optional<String> codiceFiscale,
@RequestParam Optional<String> comuneNascita,
@RequestParam Optional<String> provinciaNascita,
@RequestParam Optional<String> statoNascita) {
return personaService.cercaPersone(nome, cognome, codiceFiscale, comuneNascita, provinciaNascita, statoNascita);
}
@GetMapping("/{id}")
public ResponseEntity<PersonaDtoOutput> getPersona(@PathVariable Long id) {
PersonaDtoOutput personaDto = personaService.getPersonaById(id);
if (personaDto == null) {
return ResponseEntity.notFound().build();
}
return ResponseEntity.ok(personaDto);
}
@PostMapping
public ResponseEntity<PersonaDtoOutput> creaPersona(@RequestBody @Valid PersonaDtoInput input) {
PersonaDtoOutput created = personaService.creaPersona(input);
return ResponseEntity.status(HttpStatus.CREATED).body(created);
}
@PutMapping("/{id}")
public PersonaDtoOutput aggiornaPersona(@PathVariable Long id, @RequestBody PersonaDtoInput input) {
return personaService.aggiornaPersona(id, input);
}
@DeleteMapping("/{id}")
public ResponseEntity<Void> eliminaPersona(@PathVariable Long id) {
personaService.eliminaPersona(id);
return ResponseEntity.noContent().build();
}
}

*Questo controller espone API REST per operazioni CRUD su persone, delegando la logica al PersonaService.*

#### Service Layer per la gestione di Persona

@Service
public class PersonaService {
private final PersonaRepository personaRepository;
private final PersonaMapper personaMapper;
public PersonaService(PersonaRepository personaRepository, PersonaMapper personaMapper) {
this.personaRepository = personaRepository;
this.personaMapper = personaMapper;
}
public List<PersonaDtoOutput> cercaPersone(Optional<String> nome, Optional<String> cognome, Optional<String> codiceFiscale,
Optional<String> comuneNascita, Optional<String> provinciaNascita, Optional<String> statoNascita) {
List<Persona> persone = personaRepository.findByCriteria(
nome.orElse(null), cognome.orElse(null), codiceFiscale.orElse(null),
comuneNascita.orElse(null), provinciaNascita.orElse(null), statoNascita.orElse(null));
return personaMapper.toDtoList(persone);
}
public PersonaDtoOutput getPersonaById(Long id) {
return personaRepository.findById(id)
.map(personaMapper::toDtoOutput)
.orElse(null);
}
public PersonaDtoOutput creaPersona(PersonaDtoInput input) {
Persona entity = personaMapper.toEntity(input);
Persona saved = personaRepository.save(entity);
return personaMapper.toDtoOutput(saved);
}
public PersonaDtoOutput aggiornaPersona(Long id, PersonaDtoInput input) {
Persona persona = personaRepository.findById(id)
.orElseThrow(() -> new EntityNotFoundException("Persona non trovata"));
personaMapper.updateEntityFromDto(input, persona);
Persona updated = personaRepository.save(persona);
return personaMapper.toDtoOutput(updated);
}
public void eliminaPersona(Long id) {
personaRepository.deleteById(id);
}
}

*Il servizio incapsula la logica di business e utilizza il mapper per convertire tra DTO e entità.*

#### Repository Layer con Spring Data JPA

public interface PersonaRepository extends JpaRepository<Persona, Long> {
@Query("SELECT p FROM Persona p WHERE " +
"(:nome IS NULL OR p.nome LIKE %:nome%) AND " +
"(:cognome IS NULL OR p.cognome LIKE %:cognome%) AND " +
"(:codiceFiscale IS NULL OR p.codiceFiscale = :codiceFiscale) AND " +
"(:comuneNascita IS NULL OR p.luogoNascita.comune LIKE %:comuneNascita%) AND " +
"(:provinciaNascita IS NULL OR p.luogoNascita.provincia = :provinciaNascita) AND " +
"(:statoNascita IS NULL OR p.luogoNascita.stato = :statoNascita)")
List<Persona> findByCriteria(String nome, String cognome, String codiceFiscale,
String comuneNascita, String provinciaNascita, String statoNascita);
}

*Query personalizzata per ricerca flessibile di persone con parametri opzionali.*

#### Gestione indirizzi tramite DTO e servizio

@RestController
@RequestMapping("/api/persone")
public class PersonaController {
private final IndirizzoService indirizzoService;
public PersonaController(IndirizzoService indirizzoService) {
this.indirizzoService = indirizzoService;
}
@PutMapping("/{id}/residenza")
public ResponseEntity<IndirizzoDto> aggiornaResidenza(@PathVariable Long id, @RequestBody @Valid IndirizzoDto indirizzoDto) {
IndirizzoDto updated = indirizzoService.aggiornaResidenza(id, indirizzoDto);
return ResponseEntity.ok(updated);
}
}
@Service
public class IndirizzoService {
private final PersonaRepository personaRepository;
public IndirizzoService(PersonaRepository personaRepository) {
this.personaRepository = personaRepository;
}
public IndirizzoDto aggiornaResidenza(Long personaId, IndirizzoDto indirizzoDto) {
Persona persona = personaRepository.findById(personaId)
.orElseThrow(() -> new EntityNotFoundException("Persona non trovata"));
persona.setIndirizzoResidenza(indirizzoDto.toEntity());
personaRepository.save(persona);
return IndirizzoDto.fromEntity(persona.getIndirizzoResidenza());
}
}

*Aggiornamento dell’indirizzo di residenza di una persona tramite DTO e servizio.*

#### Gestione errori con eccezioni custom e handler globale

@RestController
@RequestMapping("/api/users")
public class UserController {
@GetMapping("/{id}")
public ResponseEntity<UserDTO> getUser(@PathVariable String id) {
if (id == null || id.isEmpty()) {
throw new BusinessException(ErrorCode.INVALID\_USER\_ID);
}
UserDTO user = userService.findById(id);
return ResponseEntity.ok(user);
}
}
@RestControllerAdvice
public class GlobalExceptionHandler {
@ExceptionHandler(BusinessException.class)
public ResponseEntity<ErrorResponse> handleBusinessException(BusinessException ex) {
ErrorCode error = ex.getErrorCode();
ErrorResponse response = new ErrorResponse(error.getCode(), error.getMessage(), error.getHttpStatus().value());
return new ResponseEntity<>(response, error.getHttpStatus());
}
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {
ErrorResponse response = new ErrorResponse("INTERNAL\_ERROR", "Errore interno del server", 500);
return new ResponseEntity<>(response, HttpStatus.INTERNAL\_SERVER\_ERROR);
}
}

*Gestione centralizzata degli errori REST con risposte JSON standardizzate.*

### 5.3 Flusso Integrato Tipico

Note: Un tipico flusso di richiesta in questa applicazione segue questi passi:

* Il client invia una richiesta HTTP (es. POST /api/persone) con un payload JSON che rappresenta PersonaDtoInput.
* Il controller REST (PersonaController) riceve la richiesta, valida i dati e chiama il metodo creaPersona del PersonaService.
* Il servizio converte il DTO in entità tramite PersonaMapper, esegue eventuali controlli di business, e salva l’entità tramite PersonaRepository.
* Il repository interagisce con il database (es. H2 o altro RDBMS) per persistere i dati.
* Il servizio converte l’entità salvata in PersonaDtoOutput e la restituisce al controller.
* Il controller risponde al client con lo status HTTP 201 Created e il DTO di output.
* In caso di errori (es. entità non trovata, dati duplicati), il servizio lancia eccezioni custom.
* Queste eccezioni sono intercettate da un handler globale che restituisce risposte di errore coerenti con codice e messaggio.

Questo flusso si applica anche ad altre entità come Indirizzo, Documento e Luogo, con controller e servizi dedicati che seguono lo stesso schema.

### 5.4 Contesto Prerequisito

Per utilizzare correttamente gli esempi sopra è necessario:

* **Spring Boot** configurato con dipendenze per Web, JPA, Validation e H2 (o altro DB).
* Annotazioni come @RestController, @Service, @Repository per la definizione dei bean.
* DTO annotati con @Valid e validazioni JSR-380 (es. @NotNull, @Size).
* Mapper implementati con MapStruct o manualmente per convertire tra DTO ed entità.
* Configurazione di sicurezza e CORS (es. CorsConfig) per abilitare chiamate da frontend esterni.
* Gestione delle eccezioni tramite @ControllerAdvice per uniformare le risposte di errore.
* File di configurazione application.properties o application.yml per definire datasource, prefissi API e limiti di upload file.
* Maven o Gradle per la build e l’esecuzione (mvn spring-boot:run).

Con questa base, gli esempi forniti sono immediatamente utilizzabili e rappresentano flussi reali di un’applicazione enterprise basata su Spring Boot.

## 6) Exception Handling

### 6.1 Mappa delle Eccezioni Custom

Nel progetto sono utilizzate principalmente le seguenti eccezioni custom:

* **BusinessException**
  + Utilizzata per segnalare errori di logica di business, come l’assenza di una persona con un ID specificato (ErrorCode.PERSONA\_NOT\_FOUND).
  + Viene lanciata da servizi come IndirizzoService, PersonaServiceImpl, DocumentoService e LuogoService quando si tenta di operare su entità inesistenti.
  + I client devono intercettarla per restituire risposte HTTP appropriate (es. 404 Not Found).
  + Non implementa meccanismi di fallback o recovery interni.
* **NotFoundException**
  + Indica che una risorsa non è stata trovata.
  + Gestita globalmente tramite @ControllerAdvice per restituire HTTP 404 con messaggi chiari.
  + Non prevede fallback o retry.
* **DuplicateEntityException**
  + Lanciata in caso di violazioni di vincoli di unicità (es. duplicazione di codice fiscale).
  + Deve essere gestita per fornire feedback chiari all’utente.
  + Non dispone di meccanismi di recovery automatici.
* **ValidationException**
  + Utilizzata per segnalare dati di input non validi, tipicamente in combinazione con annotazioni di validazione (@Valid).
  + Gestita a livello di controller o tramite handler globali.
* **Eccezioni standard di Spring e JPA**
  + EntityNotFoundException: quando un’entità non è presente nel database.
  + DataIntegrityViolationException: per violazioni di vincoli di integrità dati.
  + Gestite preferibilmente a livello di servizio o controller per tradurre in risposte API appropriate.

### 6.2 Strategia Globale di Gestione Errori

La gestione globale delle eccezioni si basa su:

* **@ControllerAdvice**
  + Centralizza la gestione di eccezioni come BusinessException, NotFoundException, ValidationException e altre eccezioni generiche.
  + Converte le eccezioni in risposte HTTP standardizzate, spesso utilizzando la classe ErrorResponse per uniformare il formato JSON degli errori.
  + Esempio tipico: mappare NotFoundException a HTTP 404 con messaggio esplicativo.
* **Handler specifici per eccezioni custom**
  + Permettono di associare codici di errore e messaggi precisi, migliorando la chiarezza per i client REST.
* **Gestione locale tramite try-catch**
  + Limitata e utilizzata solo in casi particolari, mentre la preferenza è data alla gestione globale per mantenere coerenza e ridurre duplicazioni.
* **Validazione tramite annotazioni (@Valid)**
  + Gli errori di validazione vengono intercettati automaticamente da Spring e gestiti tramite handler globali, restituendo errori 400 Bad Request.

### 6.3 Logica di Recovery e Fallback

Nel codice analizzato non sono presenti meccanismi automatici di fallback, retry o resilienza integrati come:

* **Nessun utilizzo di librerie di resilienza** quali Resilience4j o Spring Retry.
* **Assenza di fallback automatici** in caso di errori di persistenza o chiamate a servizi esterni.
* Le strategie di recovery sono demandate ai livelli superiori (controller o client) o devono essere implementate manualmente se necessario.
* Per errori specifici come superamento dimensione upload (MaxUploadSizeExceededException), si raccomanda di gestire l’eccezione a livello applicativo per fornire feedback utente.

### 6.4 Raccomandazioni per gli Sviluppatori

Per garantire una gestione coerente e robusta delle eccezioni, si consiglia di:

* **Lanciare eccezioni custom significative**
  + Utilizzare BusinessException con codici ErrorCode precisi per errori di business.
  + Evitare di usare eccezioni generiche o Exception per casi specifici.
* **Gestire le eccezioni a livello globale**
  + Implementare un @ControllerAdvice centralizzato per uniformare le risposte di errore e ridurre duplicazioni.
  + Mappare correttamente le eccezioni custom in risposte HTTP con codici e messaggi chiari.
* **Evitare catch silenziosi o troppo generici**
  + Non catturare eccezioni senza gestirle o loggarle.
  + Evitare di usare catch generici su Exception senza una logica di gestione o rilancio.
* **Validare sempre i dati in ingresso**
  + Usare annotazioni di validazione (@Valid, @NotNull, @Size, ecc.) sui DTO per prevenire errori a monte.
  + Gestire gli errori di validazione tramite handler globali per restituire messaggi utili.
* **Propagare le eccezioni quando appropriato**
  + Nei servizi, rilanciare eccezioni custom per segnalare errori rilevanti ai livelli superiori (controller).
  + Non nascondere errori di persistenza o di business.
* **Documentare i codici di errore e i messaggi associati**
  + Mantenere aggiornato l’enum ErrorCode per facilitare la comprensione e la gestione degli errori.
* **Non implementare fallback o retry senza una strategia chiara**
  + Se necessario, integrare librerie di resilienza esterne e documentare le logiche di recovery.

Seguendo queste linee guida, il progetto mantiene una gestione delle eccezioni chiara, coerente e facilmente manutenibile, facilitando il debugging e migliorando la resilienza complessiva del sistema.

## 7) Best Practices & Improvements

### 7.1 Code Quality & Design

* **Utilizzo coerente di DTO:** Impiegare DTO per isolare il modello di dominio dalla rappresentazione esterna, facilitando la validazione, la serializzazione e prevenendo l’esposizione diretta delle entità. Separare DTO di input da quelli di output per una maggiore chiarezza e sicurezza.
* **Validazione robusta:** Integrare annotazioni JSR-380 (es. @NotNull, @Size, @Email, @Pattern) sui DTO per garantire l’integrità dei dati in ingresso. Utilizzare @Valid nei controller e servizi per attivare la validazione automatica. Considerare validazioni custom o gruppi di validazione per regole più complesse.
* **Dependency Injection tramite costruttore:** Preferire l’iniezione delle dipendenze tramite costruttore (es. @RequiredArgsConstructor) rispetto a @Autowired sui campi per migliorare testabilità, immutabilità e chiarezza del codice.
* **Separazione delle responsabilità:** Mantenere la logica di business nel service layer, evitando di inserirla in controller o repository. Utilizzare pattern come Specification per ricerche dinamiche e flessibili, migliorando la manutenibilità e la leggibilità.
* **Mapping automatico:** Usare librerie come MapStruct per convertire tra entità e DTO, riducendo il codice boilerplate e minimizzando errori di conversione. Documentare mapping complessi o personalizzati per facilitare la manutenzione.
* **Gestione centralizzata delle eccezioni:** Implementare un handler globale con @ControllerAdvice per uniformare la gestione degli errori, restituire risposte API coerenti e fornire messaggi chiari e dettagliati.
* **Documentazione:** Mantenere aggiornati commenti Javadoc e, se presenti, annotazioni Swagger/OpenAPI per migliorare la comprensione e la comunicazione delle API.

### 7.2 Performance Optimizations

* **Ottimizzazione delle query:** Evitare problemi di performance come N+1 query utilizzando fetch type appropriati (LAZY vs EAGER), query personalizzate con @Query o Criteria API, e paginazione/ordinamento tramite PagingAndSortingRepository.
* **Caching:** Integrare caching con Spring Cache (es. Redis, EHCache) per ridurre il carico sul database in operazioni di lettura frequenti e dati poco volatili, migliorando la scalabilità.
* **Paginazione e filtraggio:** Implementare paginazione e ordinamento nelle API di ricerca (es. cercaPersone) per gestire grandi volumi di dati in modo efficiente e prevenire sovraccarichi di memoria.
* **Profiling e monitoring:** Utilizzare strumenti come Spring Boot Actuator, JProfiler o YourKit per monitorare le performance, identificare colli di bottiglia e ottimizzare l’uso di CPU e memoria.
* **Asincronia e reactive:** Valutare l’introduzione di metodi asincroni o reactive (es. WebFlux) per migliorare la scalabilità e la reattività dell’applicazione, soprattutto in scenari ad alto carico.

### 7.3 Security Improvements

* **Validazione e sanitizzazione input:** Applicare validazioni rigorose sui dati in ingresso per prevenire vulnerabilità come SQL Injection o XSS. Usare annotazioni di validazione e custom validators per assicurare dati coerenti e sicuri.
* **Autenticazione e autorizzazione:** Integrare meccanismi robusti come JWT o OAuth2 tramite Spring Security per proteggere le API e limitare l’accesso alle risorse in base ai ruoli.
* **Gestione CORS:** Limitare le origini consentite a quelle strettamente necessarie, evitando wildcard (\*) in produzione. Configurare dinamicamente le origini tramite file di configurazione o database e abilitare solo i metodi HTTP effettivamente utilizzati.
* **Protezione dati sensibili:** Utilizzare hashing sicuro per password, crittografia per dati sensibili e configurazioni sicure per proprietà sensibili (es. password nei file di configurazione tramite sistemi di gestione segreti).
* **Gestione centralizzata degli errori:** Evitare di esporre dettagli tecnici nelle risposte API, fornendo messaggi di errore generici ma utili, e loggare internamente le eccezioni per il monitoraggio.

### 7.4 Development Practices

* **Versionamento e branching:** Adottare strategie di controllo versione consolidate come GitFlow o trunk-based development per facilitare il lavoro collaborativo e la gestione delle release.
* **Code review:** Definire linee guida chiare per le revisioni di codice, assicurando che ogni modifica venga verificata per qualità, sicurezza e conformità agli standard.
* **Testing multilivello:** Implementare test unitari (JUnit, Mockito), di integrazione e di accettazione (Testcontainers) per garantire la correttezza funzionale e la stabilità del sistema.
* **CI/CD:** Automatizzare build, test e deploy con pipeline CI/CD (GitHub Actions, GitLab CI, Jenkins, ArgoCD) per ridurre errori manuali e accelerare il rilascio.
* **Logging strutturato:** Integrare logging dettagliato e strutturato (es. con SLF4J, Logback) per tracciare operazioni, errori e performance, facilitando il debugging e il monitoraggio in produzione.
* **Documentazione API:** Mantenere aggiornata la documentazione OpenAPI/Swagger, includendo dettagli su validazioni, errori e versioning per facilitare l’integrazione con client esterni.

### 7.5 Refactoring Opportunities

* **Semplificazione metodi complessi:** Identificare metodi troppo lunghi o con responsabilità multiple e suddividerli in funzioni più piccole e leggibili, migliorando la manutenibilità e la testabilità.
* **Modularizzazione classi:** Ridurre la dimensione di classi troppo grandi separando le responsabilità in componenti o servizi dedicati, facilitando la comprensione e la riusabilità.
* **Rimozione duplicazioni:** Individuare codice duplicato, specialmente nelle conversioni DTO-entità o nelle validazioni, e astrarre in utility o servizi condivisi per ridurre errori e facilitare modifiche future.
* **Separazione interfacce e implementazioni:** Definire interfacce per i servizi e mantenere le implementazioni separate per migliorare la testabilità e la flessibilità del codice.
* **Introduzione di builder e immutabilità:** Per DTO complessi o in crescita, valutare l’adozione del builder pattern e classi immutabili per migliorare la sicurezza, la chiarezza e la prevedibilità degli oggetti.
* **Gestione transazioni:** Annotare esplicitamente i metodi che modificano dati con @Transactional per garantire atomicità e coerenza, evitando effetti collaterali indesiderati.

Questa guida sintetizza le principali aree di miglioramento individuate nel progetto, fornendo indicazioni concrete e strumenti per elevare la qualità, la sicurezza e la scalabilità del sistema, oltre a ottimizzare i processi di sviluppo e manutenzione.

## 8) Struttura del Database

### 8.1 Modellazione delle Entità e Relazioni

Il sistema è strutturato attorno a quattro entità principali, tutte annotate con JPA per la persistenza su database relazionale:

* **Persona**
  + Mappata sulla tabella persone tramite @Entity e @Table(name = "persone").
  + Identificata da una chiave primaria id di tipo Long con generazione automatica (@Id, @GeneratedValue).
  + Contiene attributi anagrafici (nome, cognome, codice fiscale, ecc.) e riferimenti a entità correlate.
  + Relazioni:
    - @ManyToOne verso Luogo per il luogo di nascita, con chiave esterna luogo\_nascita\_id.
    - @ManyToOne verso Indirizzo per residenza e domicilio (residenza\_id, domicilio\_id).
    - @ManyToOne o @OneToOne verso Documento per i dati documentali (documento\_id).
  + Le associazioni sono gestite tramite @JoinColumn e cascade di tipo CascadeType.ALL per propagare operazioni di persistenza.
  + Non sono esplicitati indici o vincoli unici, ma si presume la presenza di vincoli di integrità referenziale e validazioni tramite annotazioni come @NotNull e @Size.
* **Indirizzo**
  + Annotata con @Entity, mappata su tabella indirizzo (nome di default).
  + Chiave primaria id di tipo Long con generazione automatica.
  + Attributi tipici: via, numero civico, CAP, comune, provincia, stato.
  + Non sono definite relazioni verso altre entità, ma è referenziata da Persona per residenza e domicilio.
  + Validazioni e vincoli sono gestiti a livello di entità, sebbene non esplicitati nel codice.
* **Luogo**
  + Entità JPA con tabella dedicata, identificata da id (Long).
  + Attributi: comune, provincia, stato.
  + Non presenta relazioni inverse o collegate, ma è referenziata da Persona come luogo di nascita.
  + Validazioni di integrità e vincoli sono implementabili tramite annotazioni JPA.
* **Documento**
  + Entità persistente con tabella omonima.
  + Chiave primaria id generata automaticamente.
  + Attributi: tipo documento, numero, ente rilasciante, data rilascio, data scadenza.
  + Non sono presenti relazioni verso altre entità, ma è collegata a Persona.
  + Possibile presenza di indici su campi di ricerca frequente (tipo, numero).

Le entità sono progettate per mantenere integrità referenziale tramite chiavi esterne e relazioni JPA, con un modello orientato a rappresentare una persona completa di dati anagrafici, indirizzi, luogo di nascita e documenti associati.

### 8.2 Progettazione dei Repository e Pattern di Accesso

Per ciascuna entità principale è definito un repository Spring Data JPA che estende JpaRepository, garantendo un accesso standardizzato e completo ai dati:

* **PersonaRepository**
  + Estende JpaRepository<Persona, Long>.
  + Fornisce metodi CRUD standard (save, findById, findAll, delete).
  + Include metodi custom per ricerche avanzate, ad esempio findByCriteria, che consente di filtrare persone in base a più parametri, inclusi attributi complessi come il luogo di nascita (con proprietà annidate).
  + Non sono presenti query di scrittura personalizzate o annotazioni @Query specifiche per operazioni particolari.
* **IndirizzoRepository**
  + Estende JpaRepository<Indirizzo, Long>.
  + Offre operazioni CRUD standard senza metodi custom evidenziati.
  + L’accesso è diretto e semplice, senza query complesse o personalizzate.
* **LuogoRepository**
  + Estende JpaRepository<Luogo, Long>.
  + Supporta CRUD standard.
  + Potenzialmente estendibile con query personalizzate per ricerche geografiche o filtri specifici, anche se non esplicitate.
* **DocumentoRepository** (presunto, non esplicitamente mostrato)
  + Probabilmente estende JpaRepository<Documento, Long>.
  + Gestisce CRUD standard per i documenti associati alle persone.

I repository seguono il pattern di delegare la gestione delle transazioni e la costruzione delle query a Spring Data JPA, sfruttando metodi derivati e, dove necessario, query JPQL o SQL personalizzate. La separazione tra DTO e entità consente inoltre di mantenere pulita la logica di persistenza e di facilitare la validazione e la sicurezza.

### 8.3 Gestione delle Transazioni

La gestione delle transazioni nel sistema è implicitamente affidata a Spring Data JPA e al livello di servizio:

* I metodi di persistenza sono tipicamente annotati con @Transactional a livello di servizio (es. PersonaServiceImpl), garantendo coerenza e atomicità delle operazioni.
* Non sono evidenziati livelli di isolamento o propagazione specifici, pertanto si assume la configurazione predefinita di Spring (propagazione REQUIRED, isolamento DEFAULT).
* Non sono presenti meccanismi espliciti di locking ottimistico o pessimista (@Version o simili), suggerendo che la concorrenza è gestita a livello di database o tramite la configurazione standard di JPA.
* Le operazioni di lettura e scrittura sono separate tra repository e servizi, con i servizi che orchestrano le transazioni e le logiche di business.
* La configurazione del database H2 in-memory con ddl-auto: update facilita lo sviluppo e i test, mantenendo la gestione transazionale trasparente e semplificata.

### 8.4 Suggerimento per Diagramma dello Schema del Database

Nota: Lo schema logico del database può essere rappresentato come segue:

* Entità principali:
  + Persona (id, nome, cognome, codice fiscale, ecc.)
  + Indirizzo (id, via, numero civico, CAP, comune, provincia, stato)
  + Luogo (id, comune, provincia, stato)
  + Documento (id, tipo, numero, rilasciatoDa, dataRilascio, dataScadenza)
* Relazioni chiave:
  + Persona ha una relazione molti-a-uno con Luogo (luogo di nascita).
  + Persona ha due relazioni molti-a-uno con Indirizzo (residenza e domicilio).
  + Persona ha una relazione uno-a-uno o molti-a-uno con Documento.
* Cardinalità:
  + Ogni Persona può avere un solo luogo di nascita, uno o più indirizzi (residenza e domicilio distinti), e un documento associato.
  + Indirizzo, Luogo e Documento sono entità indipendenti ma collegate tramite chiavi esterne.
* Raggruppamento logico:
  + Dominio anagrafico: Persona come entità centrale.
  + Dati di localizzazione: Indirizzo e Luogo.
  + Dati documentali: Documento.

Questa struttura consente una gestione modulare e scalabile dei dati anagrafici, facilitando query complesse e mantenendo integrità referenziale tra le entità.

## 9) Descrizione delle API

### 9.1 Panoramica degli Endpoint

L’architettura REST dell’applicazione è organizzata attorno a diversi controller che espongono endpoint raggruppati per ambito funzionale, con un prefisso globale configurabile tramite la proprietà ${app.api.prefix} (tipicamente /api/java). Di seguito la struttura degli endpoint principali, organizzati per controller:

* **PersonaController** (${app.api.prefix}/persone):
  + GET /search: ricerca persone con parametri di filtro opzionali, restituisce lista di PersonaDtoOutput.
  + GET /{id}: recupera una persona per ID, restituisce PersonaDtoOutput.
  + POST /form: crea una nuova persona, accetta PersonaDtoInput in JSON.
  + PUT /{id}: aggiorna una persona esistente, con ID in path e PersonaDtoInput nel body.
  + DELETE /{id}: elimina una persona per ID.
* **DocumentoController** (esempio: ${app.api.prefix}/documenti):
  + PUT /{idPersona}: aggiunge o aggiorna il documento associato a una persona, con DocumentoDto nel body.
* **LuogoController** (esempio: ${app.api.prefix}/luoghi):
  + PUT /{personaId}/nascita: aggiorna il luogo di nascita di una persona, con LuogoDto nel body.
* **IndirizzoController** (esempio: ${app.api.prefix}/indirizzi):
  + PUT /{personaId}/residenza: aggiorna l’indirizzo di residenza di una persona, con IndirizzoDto nel body.
  + PUT /{personaId}/domicilio: aggiorna l’indirizzo di domicilio di una persona, con IndirizzoDto nel body.

Questi controller utilizzano servizi dedicati (PersonaService, DocumentoService, LuogoService, IndirizzoService) per la logica di business e l’accesso ai dati, ma tali servizi non espongono direttamente endpoint REST.

Tutti gli endpoint supportano il content-type application/json per input e output. Le risorse sono identificate principalmente tramite path variables (es. {id}, {personaId}) e parametri di query (per ricerche).

### 9.2 Modelli di Richiesta e Risposta

Le API utilizzano DTO (Data Transfer Object) per incapsulare i dati di input e output, garantendo separazione tra modello di dominio e rappresentazione REST:

* **PersonaDtoInput**: modello di input per creazione e aggiornamento di persone, con annotazioni di validazione (@Valid) per garantire integrità dei dati.
* **PersonaDtoOutput**: modello di output per rappresentare i dati di una persona nelle risposte.
* **DocumentoDto**: utilizzato per rappresentare documenti associati a persone, sia in input che in output.
* **LuogoDto**: rappresenta dati relativi a luoghi (es. luogo di nascita), usato in richieste e risposte.
* **IndirizzoDto**: incapsula dati di indirizzi (residenza, domicilio), utilizzato nei payload JSON.
* **ErrorResponse**: struttura standardizzata per risposte di errore, contenente codice errore, messaggio e stato HTTP.

La validazione dei DTO è gestita tramite annotazioni di Bean Validation (@Valid) applicate nei controller, assicurando che i dati in ingresso rispettino i vincoli definiti. Le risposte di successo tipicamente restituiscono status HTTP 200 (OK) o 204 (No Content per DELETE), mentre gli errori comuni sono rappresentati con 400 (Bad Request), 404 (Not Found) e 500 (Internal Server Error), accompagnati da ErrorResponse JSON.

Non sono presenti meccanismi di versioning espliciti nelle classi DTO, ma il prefisso API può essere utilizzato per versionare gli endpoint. Caching e rate limiting non sono implementati a livello di DTO o controller.

### 9.3 Sicurezza e Autorizzazione

La sicurezza delle API è gestita principalmente a livello di configurazione globale e controller, non direttamente nelle classi di servizio o DTO. Le caratteristiche principali sono:

* **Autenticazione**: presumibilmente basata su meccanismi standard come JWT o sessioni gestite da Spring Security, anche se non esplicitamente definite nel codice fornito.
* **Autorizzazione**: non sono riportate annotazioni specifiche (@PreAuthorize, @RolesAllowed) nei controller, ma si assume che l’accesso agli endpoint sensibili sia protetto tramite configurazioni di sicurezza esterne.
* **Gestione CORS**: configurata globalmente tramite CorsConfig, che applica regole di origine per tutte le richieste API, garantendo che solo origini autorizzate possano accedere alle risorse.
* **Validazione e gestione errori**: input validati tramite annotazioni @Valid e errori di business gestiti tramite eccezioni (BusinessException, NotFoundException) intercettate da handler globali che restituiscono risposte HTTP standardizzate con ErrorResponse.
* **Protezione degli endpoint**: ogni controller si assume responsabile di applicare la sicurezza necessaria, mentre i servizi si concentrano sulla logica di business senza gestire direttamente autenticazione o autorizzazione.

Non sono presenti meccanismi di rate limiting o throttling nel codice analizzato.

### 9.4 Suggerimento per Diagramma di Flusso API

Note: Un diagramma di sequenza ideale per rappresentare il flusso tipico di una chiamata API REST in questo sistema potrebbe essere descritto come segue:

1. Il client invia una richiesta HTTP (es. POST, PUT, GET) all’endpoint REST specifico, includendo path variables, parametri di query e un body JSON se previsto.
2. La richiesta arriva al **Controller** corrispondente (es. PersonaController), che:
   * Valida il payload in ingresso tramite annotazioni @Valid.
   * Gestisce l’autenticazione e autorizzazione tramite filtri o configurazioni Spring Security.
   * In caso di errori di validazione o autorizzazione, risponde con errori HTTP standardizzati.
3. Se la richiesta è valida, il controller invoca il relativo **Service** (es. PersonaService, DocumentoService), passando i DTO convertiti in entità di dominio tramite mapper dedicati (es. PersonaMapper).
4. Il servizio esegue la logica di business, interagendo con i **Repository** (es. PersonaRepository, IndirizzoRepository) per accedere o modificare i dati nel database.
5. Il servizio restituisce i dati elaborati, che il controller converte nuovamente in DTO di output tramite i mapper.
6. Il controller costruisce la risposta HTTP con il DTO serializzato in JSON e lo status code appropriato (es. 200 OK, 204 No Content).
7. In caso di eccezioni di business o errori non previsti, un handler globale intercetta l’eccezione, genera un ErrorResponse e restituisce un codice HTTP coerente (es. 404 Not Found, 400 Bad Request, 500 Internal Server Error).

Questo flusso garantisce una chiara separazione di responsabilità tra livelli, facilita la manutenzione e assicura coerenza nelle risposte API.