SOMMARIO

[ANALISI FUNZIONALE](#_Toc1301041)

[1. Introduzione](#_Toc1301042)

[1.1 Scopo del documento](#_Toc1301043)

[1.2 Definizioni e acronimi](#_Toc1301044)

[1.3 Riferimenti ad altri documenti](#_Toc1301045)

[1.4 Panoramica del sistema](#_Toc1301046)

[2. Descrizione Generale](#_Toc1301047)

[2.1 Contesto di business](#_Toc1301048)

[2.2 Obiettivi del progetto](#_Toc1301049)

[2.3 Stakeholder e utenti target](#_Toc13010410)

[2.4 Vincoli e limitazioni](#_Toc13010411)

[2.5 Dipendenze da sistemi esterni](#_Toc13010412)

[3. Requisiti Funzionali](#_Toc13010413)

[3.1 Funzionalità Gestione Anagrafica Persona](#_Toc13010414)

[3.2 Funzionalità Gestione Indirizzi (Residenza e Domicilio)](#_Toc13010415)

[3.3 Funzionalità Gestione Documenti Associati a Persona](#_Toc13010416)

[3.4 Funzionalità Gestione Luogo di Nascita](#_Toc13010417)

[3.5 Funzionalità Gestione Configurazione CORS](#_Toc13010418)

[4. Requisiti Non Funzionali](#_Toc13010419)

[4.1 Performance](#_Toc13010420)

[4.2 Sicurezza](#_Toc13010421)

[4.3 Usabilità](#_Toc13010422)

[4.4 Scalabilità](#_Toc13010423)

[4.5 Manutenibilità](#_Toc13010424)

[5. Appendici](#_Toc13010425)

[5.1 Glossario](#_Toc13010426)

[5.2 Mockup delle interfacce](#_Toc13010427)

[5.3 Diagrammi tecnici](#_Toc13010428)

# ANALISI FUNZIONALE

## 1. Introduzione

### 1.1 Scopo del documento

Il presente documento ha l’obiettivo di fornire una descrizione funzionale chiara e dettagliata del sistema di gestione anagrafica basato su tecnologia Spring Boot. È rivolto principalmente ai team di sviluppo, analisti funzionali, responsabili di progetto e stakeholder aziendali coinvolti nella realizzazione, manutenzione e utilizzo del sistema. Il documento intende tradurre le specifiche tecniche e architetturali, già analizzate nella documentazione tecnica (ATE), in un linguaggio orientato al business, evidenziando le funzionalità offerte, i flussi operativi e le interazioni principali tra i componenti del sistema. In questo modo, si facilita la comprensione complessiva del sistema e si supporta la pianificazione delle attività di sviluppo, test e rilascio.

### 1.2 Definizioni e acronimi

Nel contesto di questo documento, si utilizzano alcuni termini e acronimi fondamentali per la comprensione del sistema. Per “Persona” si intende l’entità anagrafica principale, rappresentante un individuo con attributi quali nome, cognome, codice fiscale e dati correlati. I termini “Indirizzo”, “Documento” e “Luogo” si riferiscono rispettivamente alle informazioni di residenza/domicilio, ai documenti identificativi e ai dati geografici associati alla persona. L’acronimo API (Application Programming Interface) indica l’interfaccia REST esposta dai controller per l’interazione con il sistema. DTO (Data Transfer Object) sono le classi utilizzate per il trasferimento dati tra client e server, separate dal modello di dominio. Il termine CRUD rappresenta le operazioni di creazione, lettura, aggiornamento e cancellazione dei dati. Infine, si fa riferimento a Spring Boot come framework di sviluppo principale, e a JPA (Java Persistence API) per la gestione della persistenza dei dati.

### 1.3 Riferimenti ad altri documenti

Questo documento si basa e integra le informazioni contenute nell’Analisi Tecnica (ATE), che fornisce dettagli sull’architettura, le tecnologie adottate e le specifiche di implementazione. Ulteriori riferimenti includono le specifiche di business preliminari, le guide di configurazione del sistema e la documentazione delle API REST generate tramite OpenAPI/Swagger. Tali documenti costituiscono il contesto tecnico e funzionale di riferimento per la corretta interpretazione e utilizzo del presente documento.

### 1.4 Panoramica del sistema

Il sistema si configura come una piattaforma modulare e scalabile per la gestione completa delle informazioni anagrafiche di persone, integrando dati relativi a indirizzi, documenti e luoghi di nascita. Architetturalmente, si articola in più livelli distinti: il livello di presentazione è costituito da controller REST che espongono API per l’interazione esterna; il livello di business è rappresentato da servizi che orchestrano la logica applicativa e garantiscono la coerenza dei dati; il livello di persistenza si basa su repository JPA che gestiscono l’accesso e la manipolazione dei dati nel database relazionale. La conversione tra entità di dominio e oggetti di trasferimento dati (DTO) è affidata a componenti mapper automatici, assicurando una netta separazione tra modello interno e interfaccia esterna. Il sistema prevede inoltre una gestione centralizzata delle eccezioni per uniformare le risposte di errore e una configurazione dedicata per la sicurezza e l’accessibilità delle API, inclusa la gestione delle politiche CORS. Questa struttura consente di supportare scenari tipici di registrazione, aggiornamento e consultazione di dati anagrafici in contesti enterprise, garantendo robustezza, manutenibilità e facilità di integrazione con client esterni.

## 2. Descrizione Generale

### 2.1 Contesto di business

Il sistema nasce per rispondere alla necessità di gestire in modo centralizzato e strutturato le informazioni anagrafiche di persone, includendo dati personali, indirizzi di residenza e domicilio, documenti identificativi e luoghi di nascita. Tale esigenza si inserisce in contesti applicativi dove è fondamentale mantenere dati accurati, aggiornati e coerenti, facilitando operazioni di registrazione, consultazione e aggiornamento delle informazioni anagrafiche. La soluzione proposta si rivolge a organizzazioni che richiedono un backend affidabile e scalabile, capace di supportare flussi di lavoro tipici di gestione anagrafica, con un’attenzione particolare alla separazione dei livelli applicativi e alla qualità dei dati.

### 2.2 Obiettivi del progetto

Il progetto si pone l’obiettivo di realizzare un sistema backend basato su Spring Boot che consenta la completa gestione CRUD delle entità anagrafiche e correlate, garantendo operazioni di ricerca avanzata e aggiornamenti consistenti. Si mira a fornire API REST ben strutturate e documentate, con validazione rigorosa degli input e gestione centralizzata delle eccezioni, per assicurare robustezza e facilità d’uso da parte di client esterni. Inoltre, il sistema deve essere modulare e facilmente estendibile, con un’architettura a strati che favorisca la manutenibilità e la scalabilità futura.

### 2.3 Stakeholder e utenti target

Gli stakeholder principali includono gli sviluppatori e i team IT responsabili dell’integrazione e manutenzione del sistema, nonché gli utenti applicativi che interagiscono con le API per gestire dati anagrafici. Gli utenti target sono quindi sia i sistemi client (ad esempio frontend web o applicazioni esterne) che necessitano di accedere e modificare informazioni sulle persone, sia gli amministratori che gestiscono i dati anagrafici in modo diretto o tramite interfacce dedicate. Inoltre, i responsabili della sicurezza e della governance dei dati rappresentano un gruppo di interesse per garantire conformità e protezione delle informazioni trattate.

### 2.4 Vincoli e limitazioni

Il progetto è vincolato dall’utilizzo di tecnologie specifiche quali Spring Boot e Spring Data JPA, con un database relazionale in-memory (H2) per ambienti di sviluppo e test. Non sono presenti meccanismi di caching o resilienza avanzata integrati, né funzionalità di versioning esplicito delle API. La gestione delle eccezioni è centralizzata ma non prevede strategie di fallback o retry automatici. Inoltre, la configurazione CORS è limitata a origini specifiche per motivi di sicurezza, e non sono implementati meccanismi di rate limiting o throttling. Questi vincoli definiscono un ambito funzionale chiaro e una base tecnologica consolidata, ma lasciano spazio a possibili evoluzioni future.

### 2.5 Dipendenze da sistemi esterni

Il sistema dipende da componenti esterni quali il database relazionale (attualmente configurato con H2 in-memory per sviluppo e test) e da eventuali client esterni che consumano le API REST esposte. Non sono evidenziate integrazioni dirette con altri sistemi o API esterne all’interno del progetto. La configurazione CORS è predisposta per consentire l’accesso da origini autorizzate, tipicamente frontend web, ma non sono presenti dipendenze da servizi esterni di autenticazione o autorizzazione specifici nel codice analizzato. Eventuali estensioni future potrebbero prevedere integrazioni con sistemi di sicurezza o servizi di persistenza esterni.

## 3. Requisiti Funzionali

### 3.1 Funzionalità Gestione Anagrafica Persona

#### 3.1.1 Casi d’uso principali

La funzionalità consente la gestione completa delle entità "Persona", includendo la creazione, la ricerca, la modifica e la cancellazione di dati anagrafici. Gli utenti possono cercare persone tramite filtri opzionali quali nome, cognome, codice fiscale e dati relativi al luogo di nascita. È possibile inoltre recuperare una persona specifica tramite il suo identificativo univoco. La gestione comprende anche l’aggiornamento delle informazioni collegate come indirizzi di residenza e domicilio, documenti identificativi e luogo di nascita.

#### 3.1.2 Flussi di lavoro

Il flusso tipico inizia con una richiesta HTTP indirizzata al controller REST PersonaController, che riceve i dati in ingresso sotto forma di DTO validati (PersonaDtoInput). Il controller delega al servizio PersonaService la logica di business, che utilizza i repository JPA per accedere o modificare i dati persistenti. I mapper MapStruct convertono i dati tra entità JPA e DTO per mantenere la separazione tra i livelli. In caso di errori, eccezioni personalizzate vengono lanciate e gestite centralmente da GlobalExceptionHandler. La risposta al client è fornita con DTO di output (PersonaDtoOutput) e codici HTTP appropriati.

#### 3.1.3 Moduli funzionali

I moduli coinvolti includono:

* PersonaController: espone API REST per operazioni CRUD e ricerca.
* PersonaService e PersonaServiceImpl: implementano la logica di business per la gestione delle persone.
* PersonaRepository: interfaccia Spring Data JPA per l’accesso ai dati.
* PersonaMapper: converte tra entità Persona e DTO (PersonaDtoInput, PersonaDtoOutput).
* DTO: PersonaDtoInput per input, PersonaDtoOutput per output.
* Gestione eccezioni tramite BusinessException, NotFoundException e handler globale.

#### 3.1.4 Regole di business

La creazione e l’aggiornamento di una persona richiedono la validazione dei dati in ingresso tramite annotazioni JSR-380. La ricerca supporta filtri opzionali e restituisce risultati coerenti con i criteri specificati. L’eliminazione di una persona comporta la rimozione della relativa entità dal database. Le operazioni di aggiornamento verificano l’esistenza della persona e gestiscono errori tramite eccezioni specifiche. Le associazioni con indirizzi, documenti e luoghi di nascita sono mantenute coerenti tramite relazioni JPA e cascade.

#### 3.1.5 Interfacce utente (mappa per online bms)

L’interfaccia online espone endpoint REST sotto il prefisso configurabile ${app.api.prefix}/persone. Le principali rotte sono:

* GET /search: ricerca persone con filtri opzionali.
* GET /{id}: recupero persona per ID.
* POST /form: creazione nuova persona.
* PUT /{id}: aggiornamento persona esistente.
* DELETE /{id}: eliminazione persona.

Le richieste e risposte utilizzano payload JSON con DTO validati. La validazione e la gestione degli errori sono integrate per fornire feedback immediati all’utente.

### 3.2 Funzionalità Gestione Indirizzi (Residenza e Domicilio)

#### 3.2.1 Casi d’uso principali

Questa funzionalità permette di aggiornare gli indirizzi di residenza e domicilio associati a una persona. Gli utenti possono modificare i dati relativi a via, numero civico, CAP, comune, provincia e stato per entrambe le tipologie di indirizzo.

#### 3.2.2 Flussi di lavoro

Le richieste HTTP sono gestite dal controller IndirizzoController o tramite endpoint dedicati nel PersonaController per aggiornare gli indirizzi. I dati in ingresso sono forniti come DTO IndirizzoDto validati. Il servizio IndirizzoService recupera la persona tramite PersonaRepository, aggiorna l’entità indirizzo collegata e salva le modifiche. I DTO sono convertiti in entità e viceversa tramite mapper dedicati. Eventuali errori di persona non trovata o validazione errata sono gestiti tramite eccezioni e risposte HTTP appropriate.

#### 3.2.3 Moduli funzionali

* IndirizzoController: espone endpoint REST per aggiornamento indirizzi.
* IndirizzoService: gestisce la logica di aggiornamento indirizzi associati a persone.
* PersonaRepository: per recupero persona.
* Mapper e DTO IndirizzoDto: per conversione e trasporto dati.
* Gestione eccezioni tramite EntityNotFoundException o BusinessException.

#### 3.2.4 Regole di business

L’aggiornamento degli indirizzi richiede che la persona associata esista nel sistema. I dati forniti devono rispettare le validazioni definite nei DTO. Le modifiche sono propagate tramite relazioni JPA con cascade per garantire la persistenza coerente. Non sono previsti aggiornamenti parziali: il DTO deve contenere i dati completi dell’indirizzo.

#### 3.2.5 Interfacce utente (mappa per online bms)

Gli endpoint REST sono tipicamente:

* PUT /{personaId}/residenza: aggiorna l’indirizzo di residenza.
* PUT /{personaId}/domicilio: aggiorna l’indirizzo di domicilio.

Entrambi accettano payload JSON conformi a IndirizzoDto e restituiscono il DTO aggiornato in risposta.

### 3.3 Funzionalità Gestione Documenti Associati a Persona

#### 3.3.1 Casi d’uso principali

Questa funzionalità consente di aggiungere o aggiornare i documenti identificativi associati a una persona, quali tipo documento, numero, ente rilasciante, data di rilascio e scadenza.

#### 3.3.2 Flussi di lavoro

Le richieste HTTP sono gestite dal controller DocumentoController, che riceve DTO DocumentoDto validati. Il servizio DocumentoService recupera la persona tramite repository, aggiorna o associa il documento e salva le modifiche. Mapper dedicati convertono tra DTO e entità. Errori come persona non trovata o dati non validi sono gestiti tramite eccezioni specifiche e risposte HTTP coerenti.

#### 3.3.3 Moduli funzionali

* DocumentoController: espone endpoint REST per aggiornamento documento.
* DocumentoService: implementa la logica di gestione documenti.
* PersonaRepository: per accesso alla persona.
* Mapper DocumentoMapper e DTO DocumentoDto.
* Gestione eccezioni tramite BusinessException e NotFoundException.

#### 3.3.4 Regole di business

Il documento deve essere associato a una persona esistente. I dati devono rispettare i vincoli di validazione. L’aggiornamento sovrascrive i dati esistenti del documento collegato. La persistenza è garantita tramite relazioni JPA e cascade. Non sono previsti documenti multipli per persona in questa versione.

#### 3.3.5 Interfacce utente (mappa per online bms)

Endpoint REST principali:

* PUT /{idPersona} (sotto il prefisso /documenti): aggiorna o aggiunge il documento associato a una persona.

Il payload è un JSON conforme a DocumentoDto, con risposta contenente il DTO aggiornato.

### 3.4 Funzionalità Gestione Luogo di Nascita

#### 3.4.1 Casi d’uso principali

Questa funzionalità permette di aggiornare le informazioni relative al luogo di nascita di una persona, quali comune, provincia e stato.

#### 3.4.2 Flussi di lavoro

Il controller LuogoController espone endpoint REST che ricevono DTO LuogoDto validati. Il servizio LuogoService recupera la persona, aggiorna il luogo di nascita associato e salva le modifiche. Mapper MapStruct convertono tra DTO e entità. Errori di persona non trovata o dati non validi sono gestiti tramite eccezioni e risposte HTTP standard.

#### 3.4.3 Moduli funzionali

* LuogoController: gestisce le richieste REST per il luogo di nascita.
* LuogoService: implementa la logica di aggiornamento.
* PersonaRepository: per accesso alla persona.
* Mapper LuogoMapper e DTO LuogoDto.
* Gestione eccezioni tramite BusinessException.

#### 3.4.4 Regole di business

Il luogo di nascita deve essere associato a una persona esistente. I dati devono rispettare le validazioni definite. L’aggiornamento sostituisce i dati esistenti. La persistenza è gestita tramite relazioni JPA con cascade. Non sono previsti luoghi multipli di nascita per persona.

#### 3.4.5 Interfacce utente (mappa per online bms)

Endpoint REST:

* PUT /{personaId}/nascita (sotto il prefisso /luoghi): aggiorna il luogo di nascita di una persona.

Il payload JSON rispetta la struttura di LuogoDto e la risposta contiene il DTO aggiornato.

### 3.5 Funzionalità Gestione Configurazione CORS

#### 3.5.1 Casi d’uso principali

Questa funzionalità configura le politiche Cross-Origin Resource Sharing (CORS) per abilitare richieste HTTP da origini esterne autorizzate, garantendo la sicurezza e l’accessibilità delle API REST.

#### 3.5.2 Flussi di lavoro

All’avvio dell’applicazione, la configurazione CorsConfig definisce un bean CorsFilter che imposta le regole di origine, metodi HTTP consentiti, intestazioni e credenziali. Le richieste HTTP in ingresso sono filtrate secondo queste regole, permettendo o negando l’accesso in base all’origine della chiamata.

#### 3.5.3 Moduli funzionali

* Classe di configurazione CorsConfig annotata con @Configuration.
* Bean CorsFilter configurato con regole specifiche per origini, metodi e header.
* Integrazione con il contesto Spring Boot per applicazione globale.

#### 3.5.4 Regole di business

Le origini consentite sono configurate (es. http://localhost:4200 per frontend Angular). Sono abilitati metodi HTTP comuni (GET, POST, PUT, PATCH, DELETE, OPTIONS). Le intestazioni sono aperte a tutte (\*). Le credenziali sono abilitate per supportare autenticazione cross-origin. La configurazione deve essere sicura e limitata in ambienti di produzione.

#### 3.5.5 Interfacce utente (mappa per online bms)

Non è prevista un’interfaccia utente diretta per questa funzionalità. La configurazione agisce a livello di infrastruttura backend, garantendo che le API REST esposte dai controller (PersonaController, IndirizzoController, ecc.) siano accessibili in sicurezza da client web esterni autorizzati.

Questa sezione descrive in dettaglio tutte le funzionalità documentate nel sistema, basandosi esclusivamente sulle componenti, flussi e metodi presenti nell’Acceptance Technical Environment (ATE).

## 4. Requisiti Non Funzionali

### 4.1 Performance

Il sistema è progettato per gestire operazioni CRUD e ricerche avanzate su dati anagrafici tramite un’architettura a strati basata su Spring Boot e Spring Data JPA. Le query di ricerca sono implementate con metodi personalizzati che supportano filtri opzionali, garantendo flessibilità senza compromettere l’efficienza. L’uso di repository JPA e mapper MapStruct consente di minimizzare il codice boilerplate e ottimizzare la conversione tra entità e DTO, riducendo l’overhead computazionale durante le operazioni di mapping. Sebbene non siano esplicitati tempi di risposta o throughput specifici, l’adozione di un database in-memory H2 per sviluppo e test favorisce un rapido accesso ai dati in tali ambienti. La struttura modulare e la separazione dei livelli facilitano inoltre l’ottimizzazione futura delle performance, ad esempio tramite caching o paginazione, anche se tali meccanismi non sono attualmente implementati.

### 4.2 Sicurezza

La sicurezza del sistema è gestita principalmente attraverso la configurazione globale e le politiche CORS, che limitano le origini autorizzate per le chiamate API, garantendo che solo client attendibili possano accedere alle risorse esposte. L’uso di annotazioni di validazione JSR-380 sui DTO e la gestione centralizzata delle eccezioni contribuiscono a prevenire l’inserimento di dati non validi o malevoli, riducendo il rischio di vulnerabilità legate a input non controllati. Sebbene non siano presenti dettagli espliciti su meccanismi di autenticazione o autorizzazione all’interno del codice analizzato, si presuppone che tali aspetti siano gestiti a livello di configurazione esterna o tramite integrazioni con Spring Security. La gestione degli errori evita di esporre dettagli tecnici sensibili nelle risposte, fornendo messaggi chiari ma generici, e logga internamente le eccezioni per monitoraggio e auditing.

### 4.3 Usabilità

L’interfaccia del sistema è costituita da API REST ben strutturate e documentate tramite OpenAPI/Swagger, con endpoint organizzati per risorsa e funzionalità, facilitando l’interazione da parte di sviluppatori e client esterni. L’uso di DTO distinti per input e output, corredati da annotazioni di validazione, garantisce una chiara definizione dei contratti di comunicazione e una gestione intuitiva degli errori di validazione. La separazione netta tra controller, service, repository e mapper assicura flussi di elaborazione lineari e facilmente comprensibili, migliorando la manutenibilità e la facilità d’uso da parte degli sviluppatori. Inoltre, la configurazione CORS è pensata per consentire l’accesso sicuro da frontend moderni, come applicazioni Angular, migliorando l’esperienza di integrazione.

### 4.4 Scalabilità

L’architettura modulare e stratificata del sistema, basata su componenti chiaramente separati (controller REST, service layer, repository JPA, mapper MapStruct, entità JPA), favorisce la scalabilità sia funzionale che tecnica. L’adozione di Spring Boot consente di sfruttare meccanismi di avvio rapido, gestione dei bean e configurazioni flessibili, facilitando il deployment in ambienti containerizzati o cloud. La comunicazione sincrona tramite REST HTTP e l’uso di DTO per il trasferimento dati mantengono il sistema facilmente estendibile e integrabile con altri servizi. Sebbene non siano presenti meccanismi espliciti di bilanciamento del carico o caching, la struttura a strati e la separazione delle responsabilità permettono di introdurre tali funzionalità in futuro senza impatti significativi sull’architettura esistente.

### 4.5 Manutenibilità

Il sistema è progettato con un’attenzione particolare alla manutenibilità, adottando pattern consolidati come il Service Layer per isolare la logica di business, il Repository per l’accesso ai dati e l’uso di DTO per separare il modello di dominio dalla rappresentazione esterna. L’impiego di MapStruct per il mapping automatico tra entità e DTO riduce il codice boilerplate e facilita le modifiche future. La gestione centralizzata delle eccezioni tramite un handler globale migliora la coerenza e la robustezza del sistema, semplificando il debugging e la gestione degli errori. L’iniezione delle dipendenze, prevalentemente tramite costruttore, e l’uso di annotazioni Spring e Lombok contribuiscono a un codice pulito, modulare e facilmente testabile. La configurazione CORS e le proprietà di sistema sono gestite in modo centralizzato, facilitando la modifica e l’adattamento a nuovi requisiti senza interventi invasivi. Complessivamente, la struttura a componenti ben definita e la separazione dei livelli rendono il sistema facilmente estendibile, testabile e manutenibile nel tempo.

## 5. Appendici

### 5.1 Glossario

Nel presente documento, diversi termini tecnici e acronimi sono utilizzati per descrivere l’architettura e le funzionalità del sistema. Di seguito si forniscono le definizioni principali per facilitare la comprensione.

Il termine **API (Application Programming Interface)** indica un insieme di regole e specifiche che consentono a componenti software di comunicare tra loro. Nel contesto del sistema, le API REST espongono i servizi per la gestione delle entità anagrafiche.

**DTO (Data Transfer Object)** è un pattern utilizzato per trasferire dati tra processi o livelli applicativi. I DTO nel sistema incapsulano i dati scambiati tra client e server, separando il modello di dominio dalla rappresentazione esterna.

**Entity JPA** si riferisce alle classi annotate con Java Persistence API, che rappresentano le entità persistenti mappate su tabelle di un database relazionale.

**Spring Boot** è un framework Java che facilita la creazione di applicazioni stand-alone, con configurazioni automatiche e gestione semplificata del ciclo di vita dei componenti.

**Spring Data JPA** è un modulo di Spring che semplifica l’accesso ai dati tramite repository, fornendo implementazioni automatiche di operazioni CRUD e supporto per query personalizzate.

**Hibernate** è il provider JPA utilizzato per il mapping Object-Relational Mapping (ORM) tra entità Java e tabelle di database.

**MapStruct** è un framework per la generazione automatica di mapper che convertono tra entità e DTO, riducendo il codice boilerplate.

**Lombok** è una libreria che automatizza la generazione di codice ripetitivo come getter, setter, costruttori e metodi di utilità, migliorando la leggibilità del codice.

**JSR-380 (Jakarta Validation)** è una specifica per la validazione dei dati in Java, utilizzata per garantire l’integrità dei dati in ingresso tramite annotazioni come @Valid, @NotNull, @Size.

**CORS (Cross-Origin Resource Sharing)** è un meccanismo di sicurezza che controlla le richieste HTTP provenienti da origini diverse, configurato nel sistema tramite un filtro dedicato.

**BusinessException**, **NotFoundException** e **DuplicateEntityException** sono eccezioni custom utilizzate per segnalare errori di logica di business, risorse non trovate o dati duplicati, rispettivamente.

**Swagger/OpenAPI** è uno standard per la documentazione automatica delle API REST, che facilita la comprensione e l’interazione con i servizi esposti.

**H2 Database** è un database relazionale in-memory utilizzato per sviluppo e test, configurato tramite file di proprietà.

**JpaRepository** è un’interfaccia di Spring Data che fornisce metodi standard per operazioni CRUD e query su entità persistenti.

**@RestController**, **@Service**, **@Repository** sono annotazioni Spring che definiscono rispettivamente i componenti di presentazione, logica di business e accesso ai dati.

### 5.2 Mockup delle interfacce

I mockup delle interfacce previste per il sistema riguardano principalmente le interazioni REST esposte dai controller dedicati alle entità anagrafiche. Si prevede la realizzazione di schermate di tipo dashboard per la visualizzazione e ricerca delle persone, con filtri avanzati per nome, cognome, codice fiscale e dati geografici.

Per la gestione dettagliata delle singole entità, sono attesi moduli di inserimento e aggiornamento dati, rappresentati da form di input che utilizzano i DTO di input (PersonaDtoInput, IndirizzoDto, DocumentoDto, LuogoDto). Questi form dovrebbero includere campi per dati anagrafici, indirizzi di residenza e domicilio, documenti identificativi e luogo di nascita, con validazioni in linea per garantire la correttezza dei dati.

Le interfacce di output, corrispondenti ai DTO di output, saranno utilizzate per mostrare i dettagli delle entità in modo chiaro e strutturato, con possibilità di navigazione tra le informazioni correlate (ad esempio, visualizzare il documento associato a una persona o il luogo di nascita).

I mockup dovrebbero essere collocati nel contesto di un’applicazione web o client REST, con particolare attenzione all’usabilità e alla chiarezza delle informazioni, facilitando operazioni CRUD e ricerche complesse.

### 5.3 Diagrammi tecnici

Per rappresentare efficacemente l’architettura e i flussi del sistema, si suggerisce la realizzazione di diversi diagrammi tecnici descritti di seguito.

Un **diagramma a componenti** dovrebbe illustrare i principali layer dell’applicazione: i controller REST, il service layer, i repository JPA, i mapper MapStruct e le entità JPA. Questo diagramma evidenzierà le dipendenze e le interazioni tra i componenti, mostrando come le richieste HTTP transitano dal controller fino al database e viceversa.

Un **diagramma di sequenza** è consigliato per descrivere il flusso tipico di una richiesta CRUD. Esso dovrebbe rappresentare la sequenza di chiamate che parte dal client, passa attraverso il controller che valida i dati, invoca il servizio di business, utilizza i mapper per la conversione tra DTO ed entità, interagisce con il repository per la persistenza e gestisce eventuali eccezioni tramite un handler globale, con la risposta finale al client.

Un **diagramma UML delle classi** narrativo può descrivere le relazioni tra le entità principali (Persona, Indirizzo, Documento, Luogo), evidenziando le associazioni @ManyToOne e @OneToOne, nonché le interfacce di servizio e i mapper. Questo diagramma aiuterà a comprendere la struttura dati e le responsabilità di ciascuna classe.

Infine, un **diagramma dello schema del database** dovrebbe rappresentare le tabelle corrispondenti alle entità JPA, con le chiavi primarie, le chiavi esterne e le cardinalità delle relazioni. Questo schema faciliterà la comprensione della struttura relazionale sottostante e delle dipendenze tra i dati.

Questi diagrammi, pur non essendo inclusi graficamente in questo documento, costituiscono strumenti fondamentali per la documentazione tecnica e la comunicazione tra sviluppatori e stakeholder.