POLITECNICO DI TORINO

Corso di laurea magistrale in
INGEGNERIA INFORMATICA

Sviluppo di un atlante digitale per la memorizzazione e fruizione di informazioni su edifici storici

RELATRICE:

Prof. Silvia Chiusano

CANDIDATO:

Raniero P. Pirraglia

CORRELATORI:

Prof. Alessandro Fiori

Prof. Andrea Longhi

Novembre-Dicembre 2021

Sommario

Lista Figure

Lista Tabelle

Capitolo 1

Introduzione

L'utilizzo di archivi ha sempre caratterizzato la storia dell'umanità e delle sue attività. Immagazzinare documenti, testi, libri e in generale materiale di studio può assolvere a più funzioni. Gli archivi sono stati in passato un oggetto del potere accessibile solo a chi teneva le redini del sistema. Erano dunque utilizzati per conservare documenti importanti ma sensibili che avrebbero messo in pericolo la tenuta sociale del sistema, come del resto si continua a fare ancora oggi con la segretazione di materiale di guerra. Altro ambito di primaria importanza è quello giudiziario: tutto il materiale probatorio e i documenti relativi a un processo vengono organizzati in un archivio, per consentirne la consultazione anche dopo la sentenza.

È tuttavia quello della testimonianza storica il contesto che più facilmente si associa al concetto di archivio, ed è anche quello che più di tutti apre a considerazioni di varia natura: chi è che definisce l'importanza storica di un documento? E ammesso che un documento ne abbia, continuerà ad averne in futuro? Questo processo di selezione porta a trascurare informazioni che se fossero state preservate avrebbero potuto influenzare culture successive?
A prescindere dalle risposte a queste domande è chiaro che accumulare e organizzare adeguatamente dati, documenti, libri e quant'altro è un'attività fondamentale alla costruzione di una nostra identità storica, artistica e culturale. Basti pensare alle conseguenze esiziali che eventi come le distruzioni della biblioteca di Alessandria hanno avuto sullo sviluppo della cultura globale. Lo stesso documento può continuare a produrre informazioni potenzialmente in eterno, perché a cambiare non è il suo contenuto ma il significato che gli viene attribuito, soggetto allo *zeitgesit* di un determinato periodo storico.

La gestione di un archivio fisico è però complessa per vari motivi. In primis per ragioni puramente spaziali: anni e anni di archiviazione, anche se ben strutturata, portano a un aumento vertiginoso di materiale. Questo poi deve essere conservato in modo ottimale per contrastare il naturale processo di degradazione cui è soggetto nel tempo. Non è poi sempre agevole produrre copie e duplicati di documenti particolarmente interessanti, e ciò comporta il rischio di perdere per sempre informazioni preziose.

Va anche detto che la consultazione di un archivio fisico, sebbene conservi ancora gran fascino, non è sempre possibile: la recente pandemia di COVID-19 ha mostrato i limiti di questo approccio in una società come la nostra, dominata dall'efficienza e dalla comodità delle comunicazioni digitali. Per questa ragione e per quanto detto precedentemente è in atto un lungo

percorso di digitalizzazione dell'informazione, in cui la pandemia ha svolto il compito di catalizzatore, sia in ambito dei servizi sia per quanto concerne arte e cultura.

Il passaggio alla digitalizzazione non si limita a un trasferimento di supporto ma permette di estendere enormemente le funzionalità di un archivio, di fatto trasformandolo in qualcosa di molto più ampio e complesso. Oltre a poter essere consultati più agevolmente, i documenti possono essere analizzati, aggregati e integrati con altri dati in modo tale da costruire una piattaforma che svolga al contempo il compito di archivio, atlante e perfino museo digitale.

Nello specifico il progetto sviluppato, che prende il nome di "Atlante dei palazzi comunali", si configura come una piattaforma interdisciplinare per professionisti di vari settori che ha come oggetto di studio palazzi ed edifici impiegati come sedi di governo delle comunità urbane nel Mediterraneo. L'approccio multidisciplinare si traduce in senso tecnico in un insieme di requisiti funzionali che riguardano la modellazione di informazioni eterogenee quali cartografie, modelli 3D e dati geografici. Caratteristica fondamentale è la possibilità di poter definire semplicemente, tramite una scheda di configurazione, nuove strutture dati in grado di catturare la complessità delle informazioni che è possibile associare a un palazzo: gli Elementi Descrittivi.

L'Atlante è sviluppato come sito web avente due sezioni, una pubblica e una privata. La maggior parte delle funzionalità è implementata nella parte privata, accessibile solamente previa registrazione ad amministratori di sistema e ricercatori. Nella sezione pubblica, consultabile liberamente, è possibile visualizzare una mappa che riporta tutti i palazzi inseriti nel sistema e consultare, per ciascuno di esso, una scheda descrittiva.

Il Capitolo 2 fornisce esempi di strumenti concettualmente simili, presentandone pregi ed eventualmente difetti nell'ottica di un paragone con l'Atlante digitale sviluppato; il Capitolo 3 offre una panoramica delle tecnologie utilizzate e dei motivi che hanno portato alla loro scelta; il Capitolo 4 riguarda l'architettura dell'applicazione e quindi contiene maggiori informazioni sulle entità definite, sui pattern software implementati e sugli strumenti della piattaforma; infine il Capitolo 5 mostra vari esempi di caso d'uso, mostrando il flusso di navigazione e varie sezioni dell'interfaccia grafica. In chiusura, il Capitolo 6 è dedicato a conclusioni e sviluppi futuri.

Capitolo 2

Stato dell'arte

La società moderna sta vivendo da innumerevoli decenni un processo di informatizzazione e digitalizzazione che ha interessato tutti gli ambiti dell'agire e del sapere umano. Non è raro che torni d'attualità il dibattito che vede contrapposti i sostenitori delle competenze tecnico/scientifiche e quelli della cultura umanistica. Al netto di queste discussioni, che si configurano più come passatempi che come veri e propri momenti di sintesi, è noto a ogni professionista che progresso tecnico e discipline umanistiche non sono linee parallele, che non si incontrano mai, ma mondi che si intrecciano vicendevolmente

L'informatica infatti mette a disposizione strumenti più veloci, efficienti ed accessibili rispetto a quelli utilizzati in passato per svolgere gli stessi compiti o simili; la cultura e le idee, a loro volta, contribuiscono ad espandere o eventualmente limitare i confini del progresso. È poi naturale che in un determinato periodo storico ci sia uno sbilanciamento in un senso o nell'altro, a seconda del modello di sviluppo su cui si basa la società. In ogni caso non sarà mai possibile parlare di uomo e società senza ricorrere al concetto di cultura. Parlare di cultura in senso generale non è facile: tutto ciò che caratterizza la nascita, lo sviluppo e anche la fine di una società è materiale culturale e oggetto di studio di branche tra loro diversissime. Le conoscenze tecniche e scientifiche sono dunque parte fondamentale di questo concetto. È tuttavia invalso l'utilizzo del termine "cultura" per far riferimento prevalentemente alle materie artistiche e umanistiche. La disciplina dell'architettura cattura gran parte della complessità di questa parola, in quanto concerne aspetti sia tecnici sia artistici.

È possibile caratterizzare una società in base ai documenti scritti che ci sono pervenuti, alle opere d'arte o anche agli oggetti di uso quotidiano. L'architettura rimane uno degli ambiti di interesse maggiore per il semplice fatto che racchiude in sé una miriade di punti di vista dai quali è possibile analizzare un popolo e i suoi costumi. Lo studio di un edificio può darci un'idea delle competenze tecniche, della complessità e dell'organizzazione sociale del tempo; al suo interno posso esserci opere d'arte, libri e altri manufatti di interesse. Ma anche al di là dei suoi rapporti con il contesto in cui è inserito un edificio è interessante di per sé come manufatto umano e dunque come forma d'arte. È insomma un'entità eterogenea avente una storia molto vasta che, in alcuni casi, si protrae fino al presente.

Preservare un patrimonio culturale così vasto è dunque un compito fondamentale che presenta oggi come sempre numerose sfide. Ciò che oggi costituisce il presente potrebbe divenire

un giorno appannaggio della storia; il passare del tempo e la concentrazione di sempre più persone nelle grandi città rischia di far dimenticare alcuni edifici di interesse situati in contesti meno urbanizzati; la ricerca di spazio edificabile nelle grandi città mette in pericolo palazzi o edifici con una grande storia ma che costituiscono un ostacolo alla costruzione di infrastrutture o altro. Oltre a ciò va detto che spesso edifici e monumenti vengono attaccati e distrutti proprio perché testimoni e simboli di valori che si vogliono deliberatamente affossare: la distruzione dei Buddha di Bamiyan o delle rovine di Palmira sono alcuni esempi recenti che ci hanno dimostrato quanto siamo legati alla storia non solo della nostra nazione ma dell'umanità intera.

Dopo una presentazione così generale è poi ovvio che, data la vastità della materia di studio, i professionisti operino all'interno di ambiti specifici, definiti prevalentemente da un soggetto e un periodo storico di interesse. La piattaforma presentata in questo lavoro è stata creata per studiare edifici storici che sono stati sedi di governo delle comunità urbane nel periodo medievale, principalmente nella zona del Mediterraneo Nord-Occidentale.

Gli strumenti informatici di cui oggi dispongono i professionisti per catalogare, studiare e ricercare materiale riguardante edifici sono molto vari e potenti: è possibile costruire digitalmente un modello 3D del palazzo, acquisire in modo molto preciso informazioni relative al suo stato di conservazione, aggregare i dati provenienti da fonti eterogenee, immagazzinare documenti e fotografie, abbattere le barriere linguistiche e culturali tramite traduzioni automatiche di testi e molto altro. È inoltre possibile prevedere soluzioni che consentano anche al "pubblico" di partecipare al processo di ricerca e consultazione

Non esistono software o siti web che implementino tutte le funzionalità riportate poco sopra: poiché il materiale è potenzialmente sterminato ciascuno è focalizzato su un sottoinsieme di esse.

In questo capitolo vengono riportati alcuni esempi di siti web che per funzionalità e scopi sono simili all'Atlante Digitale sviluppato in questo lavoro.

2.1 HistAntArtSI

HistAntArtSI (Historical Memory, Antiquarian Culture, Artistic Patronage: Social Identities in the Centres of Southern Italy between the Medieval and Early Modern Period) [1] è un progetto il cui obiettivo è quello di raccogliere e studiare il materiale tramite il quale importanti famiglie locali costruivano la propria storia e identità, al fine di ottenere legittimità agli occhi di re e baroni. Si occupa dunque di fonti sia archivistiche sia letterarie, di oggetti antichi, opere d'arte, edifici e dei metodi tramite cui essi divenivano strumenti funzionali all'auto narrazione. Il contesto storico è quello del Regno di Napoli tra basso medioevo e gli inizi dell'età moderna.

Il progetto è realizzato dall'Università degli Studi di Napoli Federico II con la collaborazione del Census of Antique Works of Art and Architecture Known in the Renaissance e The Warburg Institute School of Advanced Study dell'Università di Londra ed è un sito web avente due sezioni: una in cui vengono raccolti gli articoli frutto del lavoro di ricerca e un ricco database in cui vengono catalogate tutte le informazioni su cui costruire gli studi.

Nonostante il focus sia centrato su un aspetto storico piuttosto specifico il database messo a

disposizione permette di avere una visione globale dei manufatti culturali della zona di interesse e limitrofe, che nonostante le strumentalizzazioni del passato, e forse anche grazie a queste, sono ormai divenuti patrimonio culturale di tutti.

La base di dati presenta numerose entità come Famiglie, Opera d'Arte, Luogo, Edificio Antico, Libro, Manoscritto e molto altro organizzate in schede. Ciascuna di essa è strutturata come uno schedario che permette l'accesso alla singola informazione, i cui campi sono spesso collegamenti ipertestuali ad altre entità. Per ogni città è inoltre possibile consultare una scheda onnicomprensiva di tutti gli elementi del sistema legati ad essa. La navigazione e l'organizzazione delle informazioni sono dunque ben pensate e sono inoltre presenti utili strumenti per esportare i dati di interesse e svolgere ricerche su più livelli.

Punti di forza di questo sistema sono dunque l'attenta catalogazione, soggetta al controllo critico da parte di un operatore esperto che carica i dati e la facilità con cui è possibile esportare, citare, visualizzare e ricercare le informazioni. Rispetto alla nostra piattaforma le principali differenze riguardano l'assenza di riferimenti spaziali, come ad esempio l'uso di mappa, e una diversa modellazione del rapporto tra fonti ed elementi ad esse collegate.

2.2 The Medieval Kingdom of Sicily

Lo scopo di questo sito è raccogliere immagini di varia natura riguardanti monumenti, edifici e città del Sud Italia con riferimento a un periodo che va dal 1100 al 1450. Peculiarità di questa piattaforma è la natura collaborativa del progetto: le immagini possono essere caricate da figure professionali molto diverse tra loro, quali storici, artisti, architetti o anche semplici viaggiatori, previa consultazione degli amministratori del sito. Con queste premesse è naturale conseguenza che le immagini abbiano una natura eterogenea: sono presenti dipinti, disegni, rilievi, fotografie, opere d'arte e piani architettonici [2].

Questo strumento è stato dunque concepito per essere solamente una raccolta di immagini, eventualmente utilizzabile come supporto all'attività di ricerca, e non come piattaforma contenente informazioni storiche complete e complesse. Le immagini sono organizzate e indicizzate in base a città, edifici, artisti e collezioni. A seconda dell'entità usata come base della ricerca viene visualizzata una scheda contenente varie sezioni, delle quali la principale è ovviamente quella che racchiude tutte le immagini. È anche possibile consultare, dalla Home del sito, una mappa che riporta tutti i luoghi inseriti nel sistema e da lì accedere all'insieme di immagini ad essi associati.

L'idea del progetto è stata concepita nel 2008 da Caroline Bruzelius (Duke University) e William Trozo (UC, San Diego) è successivamente sviluppata dalla Duke University. Il database è stato sviluppato con il software Claris FileMaker® ed è di tipo relazionale. Il sito web utilizza tecnologie standard quali JavaScript, HTML, CSS per il frontend e PHP per il backend, che prevede l'uso di 360Works SuperContainer per la gestione delle immagini. Le mappe sono implementate con l'API di Google Maps.

In questo contesto non ha tanto senso parlare di differenze con l'Atlante Digitale sviluppato poiché sono proprio le premesse e gli obiettivi alla base dei due progetti ad essere diversissimi:

Medieval Kingdom of Sicily è stato creato per inserire e consultare immagini, l'Atlante Digitale per essere una piattaforma di studio e ricerca comprendente informazioni varie ed eterogenee. È dunque naturale che lo strumento presentato in questo paragrafo offra funzionalità interessanti come la costruzione di una iconografia storica, che potrebbe essere oggetto di uno degli sviluppi futuri dell'Atlante. Anche la partecipazione di utenti esterni progetto è sicuramente uno dei punti di forza di Medieval Kingdom. Un punto di contatto è costituito dalla presenza di una mappa che permette di consultare luoghi ed edifici inseriti nel sistema.

2.3 Mapping Gothic France

Mapping Gothic France è un sito web che offre agli utenti la possibilità di effettuare tour virtuali all'interno di edifici realizzati in stile gotico, principalmente in Francia [3]. La navigazione del sito è essenziale ed efficace: all'utente viene mostrata una mappa riportante tutti gli edifici esistenti nel sistema e da qui è possibile selezionare quello di cui si vuole consultare la scheda privata. Per ogni edificio è disponibile una serie di strumenti inerenti a due categorie: media e testo.

Per quanto concerne i media, ne esistono di vari tipi a supporto di funzionalità molto interessanti: le immagini sono raggruppate in base all'area, che è possibile cambiare da un menù a tendina o anche selezionare da una piantina, se disponibile; nella sezione *Panoramas* sono presenti immagini a 360 gradi; *VR Tour* permette di muoversi all'interno e all'esterno degli edifici; *Laser Scans* contiene scansioni 3D; *Gigapixel Images* presenta foto ad altissima definizione; *Historical Images and Plans* offre un percorso tra immagini, foto e disegni appartenenti ad epoche diverse.

Per ogni edificio sono poi disponibili informazioni di natura testuale, nello specifico riguardanti descrizioni e bibliografia. A livello funzionale questa parte non è minimamente paragonabile a quella dedicata ai media, che costituiscono l'interesse principale della piattaforma. La preponderanza delle funzionalità "visive" deriva dalla necessità, da parte dei creatori e curatori del progetto, di identificare un approccio capace di rappresentare in modo efficace lo spazio, e in particolare lo spazio Gotico, su uno schermo di un computer.

La sviluppo della piattaforma è frutto della collaborazione tra Media Center for Art History in the Department of Art History and Archaeology della Columbia University, il Visual Resources Library at Vassar College, e the Columbia University Libraries e, stando a quanto riporta la sezione *About* del sito [4] la versione di base è realizzata con tecnologia open source.

Rispetto all'Atlante Digitale, Mapping Gothic France è molto più incentrato sulla componente visiva degli edifici ed è dunque più vicino a un museo virtuale che a un atlante. Le scansioni 3D, le immagini a 360 grandi, la possibilità di selezionare le immagini in base alla posizione sulla piantina sono funzionalità molto interessanti che la nostra piattaforma, a ora, non possiede. Tuttavia la componente archivistica e testuale, sebbene presente e utile, costituisce più che altro un supporto alla fruizione degli strumenti multimediali che materiale di studio e ricerca.

2.4 Tecniche murarie tradizionali: conoscenza per la conservazione ed il miglioramento prestazionale

Obiettivo di questo progetto è realizzare un sistema informativo spaziale (SIS) in grado di immagazzinare, studiare e preservare le varie tecniche costruttive utilizzate in Sardegna nel periodo compreso tra il 13° e 18° secolo. L'approccio è multi-disciplinare è comprende il contribuito di numerosi professionisti: il sistema tratterà dunque informazioni inerenti ad architettura, scienze dei materiali, ingegneria energetica, meccanica e fisica

Il sistema informativo è sviluppato dall'Università di Cagliari ed è interamente basato su software open-source, in modo tale da offrire uno strumento di ricerca alla comunità dei professionisti del settore. Il database utilizzato è PostGIS (estensione spaziale di PostgreSQL) ed è collegato a un WebGIS (Geographical Information System) mediante un WMS (Web Map Service). Quest'ultimo è basato sull'utilizzo del plugin javascript Leaflet, usato anche nel nostro progetto.

Poiché il prodotto non è stato ancora rilasciato per una lista delle funzionalità si fa riferimento all'articolo riportato nella fonte [5]. L'entità di base è l'edificio, che è possibile esplorare in tre dimensioni. Al suo interno sono individuabili elementi che rimandano a tecniche di costruzione, materiali e molto altro. Il fine è quello di compilare una scheda in grado di caratterizzare l'edificio sotto numerosi punti di vista: stato di conservazione e conseguente possibilità di restauro, materiali utilizzati, datazione precisa e molto altro.

L'esplorazione spaziale, che il nostro Atlante non possiede, è sicuramente la caratteristica principale di questo strumento. Tuttavia il suo ambito di ricerca è molto specifico se paragonato al nostro, che prevede la presenza di fonti eterogenee che potrebbero, in futuro, includere anche rilievi, modelli 3D e in generale molte delle informazioni che *Tecniche Murarie* tratta.

2.5 Carta di Rischio

La **Carta del Rischio** è un sistema informativo territoriale (SIT) realizzato dal la Direzione Generale Sicurezza del Patrimonio culturale per fungere da supporto, sia scientifico sia amministrativo, agli enti che si occupano di tutelare il patrimonio culturale italiano [6]. Il sistema è costituito da un database contenente dati sia alfanumerici sia cartografici che vengono analizzati con approccio statistico per costruire un modello di rischio. Tramite questo è poi possibile individuare le vulnerabilità di un bene culturale.

Le funzionalità messe a disposizione dal sistema includono la possibilità di consultare e aggiornare non solo il repertorio dei beni culturali ma anche informazioni di natura legislativa, schede di vulnerabilità di beni mobili e immobili, analisi sismiche, statistiche sui beni immobili.

Il contesto di utilizzo di questo strumento è molto diverso da quello dell'Atlante, il quale condivide con la "Carta di Rischio" solamente l'utilizzo di una mappa su cui sono indicati gli oggetti di studio e la presenza di una scheda dettagliata. È stato citato perché realizzato tra il 1992 e il 1996 e dunque rappresenta uno dei primi esempi di strumenti trattanti edifici situati su mappe.

2.6 THEMAS

Thesaurus **Ma**nagement **S**ystem – THEMAS è un software open source usato per la gestione di thesauri multi-lingua. Un *thesaurus* è un database terminologico relativo a un dominio specifico. È stato sviluppato da FORTH-ICS (Foundation for Research and Technology Hellas - Institute of Computer Science) [7]

Funzionalità chiave del sistema sono gestione e amministrazione di diverse tipologie di relazioni semantiche all'interno di un thesaurus, la facilità di navigazione tra parole e lemmi in relazione tra loro, espressivi strumenti di ricerca, diverse modalità di rappresentazione grafica di concetti e del loro contesto. THEMAS è utilizzabile in diversi domini di ricerca grazie alle possibilità di configurare un vasto numero di opzioni: attivazione dei sistemi di controllo di consistenza, lingua della interfaccia grafica e del thesaurus stesso, l'insieme delle traduzioni, le componenti dell'interfaccia grafica. Il sistema permette inoltre di definire diverse tipologie di utenti, aventi diversi livelli di privilegio. È in definitiva uno strumento usato per l'analisi della struttura logica e della consistenza di database terminologici.

Passando al piano delle tecnologie, l'interfaccia grafica è implementata come applicazione web usando Java Servlets; il database usato è Neo4j, la cui informazione è strutturata come grafo; i thesauri seguono i principi del linguaggio relazionale TELOS.

Nonostante THEMAS non tratti di edifici, palazzi e località come l'Atlante Digitale le due piattaforme presentano punti di contatto a livello di requisiti funzionali: entrambi sono pensati per supportare più lingue e permettono una configurazione dinamica di elementi descrittivi. Anche il rapporto tra quest'ultimi e le fonti che li citano è simile nei due sistemi.

2.7 ResCult

RESCULT (**Resilience of Cultural heritage** ha come obiettivo quello di prevenire e mitigare l'impatto che eventi distruttivi possono avere sui patrimoni culturali attraverso la costruzione di un database europeo integrato e accessibile ai diversi livelli della amministrazione: Protezione Civile, Ministeri dei Beni Culturali, Unione Europea, Autorità locali [8]. Oltre a ciò, il sistema si propone di fornire una strategia per la riduzione del rischio attraverso l'identificazione di azioni e investimenti tali da aumentare la prevenzione e incrementare le capacità di resilienza dei siti culturali.

Il progetto è stato coordinato dall'Istituto Superiore sui Sistemi Territoriali per l'Innovazione, in collaborazione con altri enti ed università tra cui il Politecnico di Torino.

La geolocalizzazione degli edifici, la possibilità di compilare per ciascuno di essi delle schede descrittive e l'inserimento di informazioni di varia natura a opera di diversi ricercatori sono caratteristiche che accomunano ResCult al nostro Atlante. Tuttavia quest'ultimo è più incentrato sulla ricerca e lo studio di fonti storiche che sulla prevenzione del rischio. Un vantaggio che ResCult possiede è quello di poter inserire modelli 3D nel sistema, funzionalità che è comunque prevista negli sviluppi futuri dell'Atlante.

2.8 HERMES

Hermoupolis Digital Heritage Management (HERMES) è il risultato di otto anni di ricerca condotta dall'Università Nazionale Tecnica di Atherns e la Municipalità di Hermoupolis. Il contesto è locale, nel senso che oggetto di studio sono gli edifici presenti nella città di Hermoupolis.

Obiettivo primario è, come per ResCult, valutare le vulnerabilità di edifici storici e proporre un modello in grado di sostenere il processo di misurazione della priorità di intervento su di un determinato edificio. Per realizzare questo compito è stato creato un database che contiene informazioni quali la posizione nella città, lo stile architettonico, la tipologia dell'edificio, lo stato di conservazione, il bisogno di intervento, il rischio del collasso [9]. Gli edifici sono segnati su mappa, implementata con il plugin Leaflet, ed è possibile lasciare commenti per ciascuno di essi, così come proporre aggiunte e modifiche a quelli esistenti.

Anche in questo caso i punti di contatto con l'Atlante digitale sono la presenza di una mappa e la possibilità di compilare una scheda descrittiva per gli edifici. Informazioni e fonti di natura storica sono assenti o limitati.

2.9 Arches

Arches è una piattaforma open-source sviluppata dal Getty Conservation Instituite e dal World Monuments Fund per la gestione di dati relativi al patrimonio culturale. Può essere installata, configurata ed estesa liberamente a seconda dell'utilizzatore e del dominio applicativo. Il software include un sistema per la gestione dei dati, strumenti per ricerca e visualizzazione dei dati (anche geospaziali), strumenti per la gestione di progetti e task utili per procedure complicate di modifiche ai dati. [10].

La versione di base della piattaforma è configurata per gestire dati relativi a sei entità: *Heritage Resource* (edifici e siti archeologici), *Heritage Group*, *Activity* (sondaggio, intervento parziale), *Historical Event* (guerra, terremoti, frana), *Actor* (architetto, organizzazione, celebrità) e *Information Resource* (modelli 3D, foto, report). Tutte queste risorse sono liberamente modificabili e configurabili. È inoltre possibile definire relazioni tra queste entità, con l'obiettivo di seguire alcune organizzazioni o persone specifiche.

Di tutti i sistemi citati in questo capitolo, Arches è senza dubbio il più vicino alle specifiche dell'Atlante, soprattutto per quanto riguarda l'uso di entità simili. Un punto a sfavore è la mancanza della possibilità di associare elementi descrittivi a fonti.

2.10 Conclusioni

In questo capitolo sono stati mostrati sistemi e piattaforme molto diversi tra di loro, per oggetto di studio e feature implementate. La tabella [**tab:st˙art**] sintetizza pregi e difetti di ciascuno di essi. Tuttavia i requisiti dell'Atlante coprono la maggior parte delle funzionalità

presentate: configurazione di elementi descrittivi dinamici, gestione e amministrazione di utenti con vari livelli di privilegio, consultazione di una mappa, gestione delle fonti storiche, possibilità di consultazione in più lingue.

Sistema	**Pro**	**Contro**
HistAntArtSI	- Controllo critico in fase di catalogazione - Potenti strumenti di ricerca - Visualizzazione integrata dell'informazione	- Assenza di riferimenti geospaziali
The Medieval Kingdom of Sicily	- Progetto con natura collaborativa - Grande selezione di immagini, divise in base a diverse categorie - Geolocalizzazione edifici	- È sostanzialmente un database di immagini
Mapping Gothic France	- Immagini a 360 gradi - Modelli 3D - Geolocalizzazione edifici	- Focus su esperienza visiva, fonti letterarie presenti ma limitate
Tecniche Murarie	- Esplorazione 3D dei palazzi - Elementi descrittivi dinamici	- Non ancora rilasciato - Focus limitato a tecniche di costruzione
Carta di Rischio	- Geolocalizzazione edifici	- Focus limitato su prevenzione del rischio
Themas	- Data Model che cattura la relazione tra fonti ed elementi descrittivi	- Non riguarda edifici ma la gestione di thesauri
ResCult	- Modelli 3D - Geolocalizzazione edifici	- Assenza di fonti storiche
Hermes	- Geolocalizzazione edifici - Possibilità di inserire commenti	- Limitata presenza di fonti storiche
Arches	- Estendibile - Geolocalizzazione edifici - Gestione di numerose entità diverse	- Mancanza relazione tra fonti ed elementi descrittivi

Tabella 2.1: Pro e Contro del sistemi presentati

Capitolo 3

Tecnologie dell'Atlante Digitale

Nella sua definizione più generale, un Sistema Informativo (*Information System*) si occupa di gestire informazioni. Esso non è qualcosa di strettamente "fisico" ma risulta composto da persone, regole, processi e tecnologie la cui interazione reciproca e strutturata rende possibile l'erogazione di un servizio [11]. In questo caso il servizio offerto è la consultazione di un Atlante Digitale che permette la memorizzazione, l'acquisizione e la fruizione di informazioni sugli edifici storici. Poiché internet è la tecnologia principale su cui poggia per svolgere le sue funzioni, si parla di *Web Information System*.

L'informazione presenta una forma diversa a seconda del contesto e degli strumenti che la gestiscono: può essere una mail che illustra i requisiti richiesti dal committente del progetto, una collezione di documenti in un database o ancora un file JSON in una comunicazione tra frontend e backend. Nella progettazione di un sistema IT (*Information Technology*) questa varietà si traduce in un insieme di scelte da compiere, a seconda delle necessità del caso: quale piattaforma utilizzare per la distribuzione dell'applicazione? DBMS relazionale o NoSQL? È meglio utilizzare un framework per il frontend moderno e pionieristico o affidarsi a uno più vecchio ma che ha raggiunto lo stato dell'arte? Le decisioni vengono prese a fronte di una fase iniziale di design e architettura del sistema, cui partecipano tutti gli stakeholders. Il punto di partenza è spesso rappresentato da un documento, il *requirements document*, che mostra quali sono i requisiti e le funzionalità che il sistema dovrà rispettare e implementare. Segue poi una fase in cui, a partire dai requisiti appena citati, viene condotto uno studio di fattibilità che si occupa di stimare i costi del progetto, selezionare le tecnologie da utilizzare e definire una timetable per lo sviluppo.

In questo capitolo sono presentate e descritte le principali tecnologie dell'Atlante digitale.

3.1 Piattaforma di Distribuzione

Nell'introduzione a questo capitolo sono illustrate brevemente alcune delle attività iniziali del ciclo di sviluppo del software: l'analisi preliminare e la progettazione. Queste sono generalmente seguite da quelle di realizzazione e collaudo. Terminati i test, si passa alla fase di **distribuzione** (*deployment* in inglese), che consiste nel rilasciare il software e metterlo in

funzione, passando attraverso l'installazione e la configurazione dello stesso.

Generalmente l'infrastruttura hardware/software di sviluppo di un'applicazione e quella sulla quale deve essere installata ed eseguita differiscono di molto. Ciò comporta ovviamente una serie di problemi che richiedono tempo ed energia extra. Per queste ragioni sono state sviluppate tecnologie per facilitare il processo di distribuzione. Una di queste è la **virtualizzazione**, un processo che porta alla creazione di un livello di astrazione al di sopra dell'hardware. Su questo livello possono essere in esecuzione più **Virtual Machines**, ambienti virtuali che emulano computer fisici, ciascuno dotato del proprio sistema operativo. L'interazione tra VMs e hardware sottostante avviene tramite **l'hypervisor**, software che si occupa di allocare le giuste risorse a ciascuna VM, che risulta completamente isolata dalle altre [12]. L'utilizzo di VMs permette di far coincidere l'ambiente di sviluppo con quello di distribuzione ma presenta un ovvio svantaggio: i sistemi su cui viene rilasciata l'applicazione non eseguono direttamente quest'ultima ma un intero computer virtuale, operazione computazionalmente molto pesante.

Per questa ragione si ricorre sempre di più a una nuova tecnologia, naturale evoluzione delle macchine virtuali: i **container**. Un container è un ambiente di esecuzione isolato che contiene tutte le dipendenze e le componenti necessarie al funzionamento di un software. Più container indipendenti possono essere in esecuzione sulla stessa istanza del sistema operativo *linux-based* grazie alle funzionalità di isolamento che quest'ultimo mette a disposizione. In particolare il kernel Linux utilizza dei *namespace* che non consento ai container di accedere liberamente alle risorse operative del sistema, come ad esempio l'albero dei processi, i file system, la CPU o la memoria. Come conseguenza più container condivideranno sì lo stesso kernel della macchina host ma ciascuno di essi svolgerà le proprie funzioni in un ambiente operativo privato, con un numero limitato di risorse e servizi a disposizione e potendo disporre di un file system dedicato [13]. L'immagine 3.1 mostra i diversi livelli di astrazione di VMs e container.

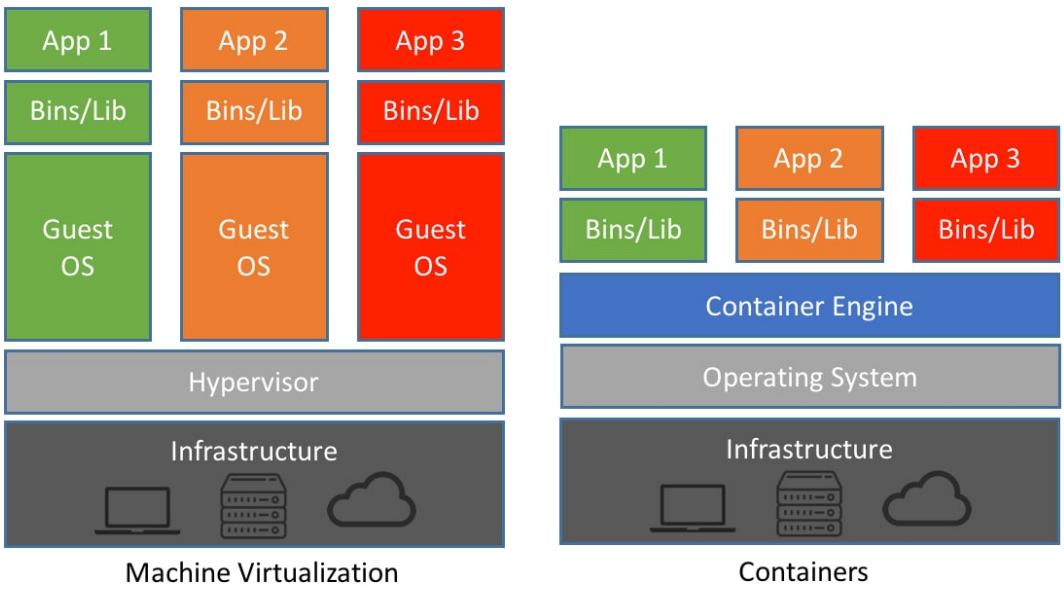

Figura 3.1: Virtual Machines vs Container (immagine dal web)

Se con le VMs l'isolamento è dunque completo perché ciascuna di esse ha il suo S.O. (e per questa ragione vengono utilizzate ancora massicciamente in ambiti in cui la sicurezza è fondamentale), nel contesto dei container si parla di isolamento lasco (*loosely* in inglese) [14]. Per la loro leggerezza e facilità di utilizzo i container sono stati preferita alle VMs come tecnologia di distribuzione per questo progetto. La piattaforma utilizzata per la containerizzazione dell'applicazione è Docker.

Docker ha un'architettura di tipo client-server: il client si occupa di formulare richieste mediante la Docker API al Docker *daemon*, il quale funge da server e svolge i compiti principali: costruire i container, guidarne l'esecuzione, consultare il registro delle immagini. Client e daemon possono comunicare in locale, utilizzando i socket UNIX, oppure in rete mediante REST API [15]. Nell'immagine 3.2 è illustrato il flusso d'esecuzione ad alto livello di tre comandi Docker molto comuni, ovvero *build*, *pull* e *run*.

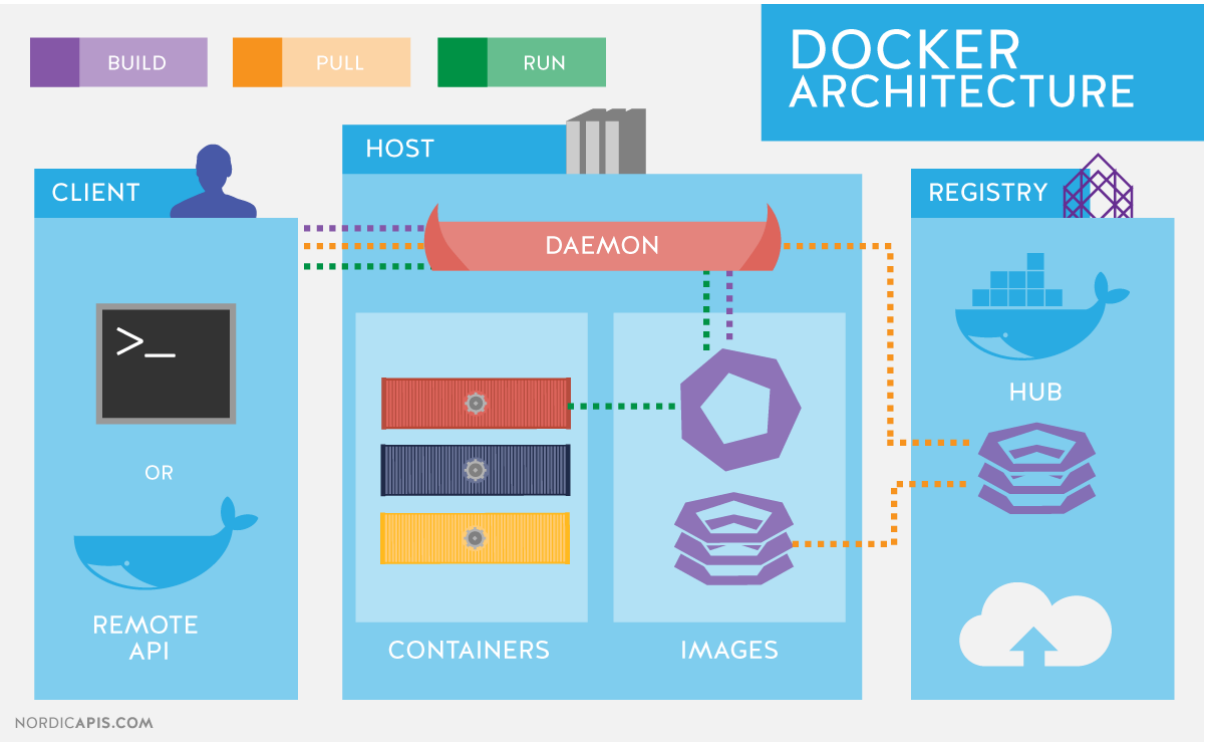

Figura 3.2: Architettura Docker [16]

Il Docker daemon lavora su entità come immagini, container e volumi la cui comprensione è necessaria per capire appieno il funzionamento della piattaforma.

Una **Docker Image** è un template in sola lettura contenente le istruzioni necessarie alla creazione di un container. L'immagine viene a sua volta costruita a partire da un **Dockerfile**, file di testo in cui sono presenti, ordinatamente, tutti i comandi da eseguire in sequenza per creare l'immagine. Quando il client invia il comando **"docker build ."** il daemon si occupa di generare un'immagine a partire dal Dockerfile e da un contesto. Il contesto non è altro che l'insieme dei file e delle cartelle di un percorso: nel caso del comando riportato poco sopra è "." ovvero la cartella corrente. Il daemon procede ad analizzare ricorsivamente il contesto e terminato il compito si occupa di validare sintatticamente il Dockerfile. Se non sono presenti errori, le istruzioni del Dockerfile vengono eseguite sequenzialmente. Va notato che ogni

istruzione è indipendente dalle altre e comporta la creazione di un nuovo livello, detto *layer*. Alla fine del processo il daemon presenterà in output l'ID dell'immagine finale, composta dai vari layer creati. Se si vuole apportare una modifica al Dockerfile il processo di ricostruzione dell'immagine interesserà solamente i layer che hanno subito un cambiamento [17]. Ottenuta l'immagine è possibile mandare in esecuzione un container, che non è altro che un'istanza eseguibile dell'immagine. Se si modifica un container in esecuzione Docker permette di trasferire i cambiamenti sull'immagine. Le immagini sono anche liberamente scaricabili e caricabili dal **Docker Hub**, un registro pubblico aperto a tutti e già integrato nel sistema Docker.

Per quanto concerne la persistenza dei dati generati e utilizzati dai container esistono due meccanismi differenti: bind mount e Volumi. La tecnica del **bind mount** consiste nel montare un file o una cartella direttamente nel container. Ciò comporta l'utilizzo di un percorso assoluto che dipende dalla struttura delle cartelle del sistema operativo della macchina host [18]. L'uso di un **Volume**, invece, porta alla creazione di una cartella all'interno di uno spazio nel file system della macchina host opportunamente dedicato a Docker. Questo spazio è gestito dalla piattaforma ed è possibile interagire con esso mediante l'uso del Docker client . I vantaggi dell'utilizzo dei volumi rispetto al bind mount includono la facilità con cui è possibile effettuare operazioni di back-up e migrazione, la compatibilità con Linux e Windows la semplicità con cui è possibile condividerli tra i vari container [19].

L'applicazione sviluppata, tuttavia, non consta di un solo container ma è basata sull'architettura a **microservice**: ogni container impacchetta un microservizio, ovvero una (relativamente) piccola componente dell'applicazione che svolge un compito preciso. L'applicazione nella sua interezza risulta dunque il frutto del lavoro e della comunicazione strutturata di tutti i microservizi che la compongono. I pregi di questa architettura sono principalmente legati all'accoppiamento lasco (*loosely coupling*) dei microservizi: poiché essi comunicano mediante interfacce (come ad esempio le REST API) la loro interdipendenza è limitata. Ciò consente di manutenere e testare ogni componente in modo facile e veloce, senza dover condurre test integrativi. Per lo stesso motivo ogni container o microservizio è singolarmente distribuibile e dunque riutilizzabile in più contesti. Un'applicazione siffatta richiede però strumenti che coordinino la comunicazione tra i vari container. A seconda della complessità del sistema e della quantità di container, si può scegliere un'architettura multi nodo, in cui i vari container vengono distribuiti su più host o una a singolo nodo, in cui è presente un solo host. Nel primo caso è necessario utilizzare strumenti per l'orchestrazione di container distribuiti, come ad esempio Kubernetes.

Kubernetes è una piattaforma piuttosto complessa che gestisce in modo automatico un cluster di host su cui sono in esecuzione container: implementa politiche di fault-tolerance, gestisce le risorse in modo ottimale e garantisce scalabilità orizzontale. Viene principalmente utilizzata in progetti molto grandi in cui la scalabilità è di fondamentale importanza [20]. Nel nostro caso la quantità di container non è tale da giustificare il ricorso a un'architettura distribuita. Per questa ragione è stato utilizzato **Docker Compose**, uno strumento più semplice che permette di gestire l'esecuzione di un'applicazione multi-container su un singolo host. La definizione e configurazione dei servizi dell'applicazione è riportata all'interno di un file in formato YAML, che Compose utilizza per svolgere le sue funzioni, ovvero avviare, fermare e ricostruire un servizio e visualizzarne stato e log di output. Compose permette inoltre di eseguire più copie isolate dello stesso ambiente su un singolo host, garantisce la persistenza

dei volumi alla creazione dei container e il riuso di quelli per i quali la configurazione non è cambiata [21].

3.2 Database Management System

Tutte le applicazioni di una certa complessità necessitano di un Database Management System, ovvero di uno sistema software la cui funzione è quella di gestire in modo efficiente una base di dati. Per molti decenni la scelta del DBMS ha riguardato esclusivamente prestazioni e funzionalità offerte ma non la struttura dei dati, che ha sempre seguito il modello relazionale.

Nei database **relazionali**, o SQL (dal nome del linguaggio utilizzato per la loro gestione e interrogazione) l'informazione è organizzata in tabelle dallo schema ben definito. Ogni tabella è formata da righe e colonne; quest'ultime costituiscono i campi descrittivi dei vari record, ovvero delle righe stesse. Ogni tupla inserita nella tabella dovrà dunque rispettarne la struttura, indicando un valore per ogni colonna (eventualmente nullo). Alcuni esempi di DBMS di tipo relazionali sono Oracle, MySQL, Microsoft SQL Server e PostgreSQL. Una struttura così rigida è giustificata dal fatto che in passato la natura dell'informazione non era così dinamica come ai nostri giorni, che hanno visto l'esplosione del web e conseguentemente la produzione di enormi quantità di dati più complessi. Oltre alla complessità sono vertiginosamente aumentati il numero di dispositivi che producono dati e la velocità di rete. In definitiva, oggi ci si ritrova a dover gestire enormi volumi di dati, strutturalmente complessi, provenienti da un gran numero di fonti eterogenee. I database relazionali non sono indicati per questo compito per varie ragioni che verranno approfondite in questa sezione.

In risposta a questi cambiamenti si è resa necessaria l'adozione di un nuovo paradigma nella gestione della natura dei dati. Sono dunque nati i database **non-relazionali**, o NoSQL, che lavorano su dati che non sono strutturati secondo uno schema predefinito. Esistono basi di dati che organizzano l'informazione sotto forma di grafo, coppie chiave-valore, documenti, colonne. Va detto che una soluzione non esclude l'altra: in un sistema molto complesso, come un social network, alcuni dati vengono naturalmente modellati come grafo (i.e. la rete di amici) ma altri possono essere organizzati in documenti ed è anche possibile che sia contemporaneamente presente anche un DBMS relazionale.

Oltre alla struttura dei dati, un altro problema dei RDBMS (la R sta per Relational) è l'architettura a singolo nodo: questo costituisce un *single point of failure* e non permette la gestione di grandi volumi di dati. Un database relazionale presenta dunque una scalabilità verticale, più costosa rispetto a quella orizzontale, offerta dai DBMS NoSQL. Si parla di scalabilità orizzontale perché, al contrario di quella verticale, che richiede di aumentare le prestazioni di un singolo server aggiungendo sempre più memoria RAM e CPU più potenti, essa è basata sulla semplice distribuzione e replicazione di dati su più nodi. Ciò rende le interrogazioni più rapide ma porta a un aumento delle dimensioni del database, a causa della presenza di repliche. Tuttavia, il costo dei dischi è molto minore se paragonato a quello di CPU e RAM.

La rigidità della schema delle tabelle e l'architettura a singolo nodo nei database SQL offrono però alcuni vantaggi: in particolare è possibile effettuare più facilmente l'operazione di

join tra tabelle e garantire le **transazioni ACID**. Una transazione è un insieme di operazioni su più tabelle o collezioni che tuttavia deve essere trattato come una singola unità di lavoro: esempio tipico è il trasferimento di denaro tra due conti bancari. In una transazione va garantito che o vengono portati a termine tutti i sotto task che la compongono oppure nessuno di essi, e in questo caso si ritorna allo stato precedente all'avvio della transazione senza lasciare il sistema in uno stato inconsistente. Una transazione che possa essere definita affidabile deve seguire le quattro proprietà ACID, ovvero Atomicità, Consistenza, Isolamento e Durabilità. Garantire queste proprietà richiede però un costo che i NoSQL, per costruzione, non sono disposti a sostenere. Ad esempio garantire la consistenza del sistema significherebbe intervenire su tutte le repliche di dati disseminate nei vari nodi del sistema, e ciò comporterebbe un grande abbassamento delle performance [22].

L'applicazione presentata non prevede la gestione di grandi volumi di dati provenienti da fonti eterogenee ma è sviluppata per lavorare su dati e entità dalla struttura molto flessibile. La scelta di un DBMS NoSQL è dunque obbligata ed è ricaduta in particolare su MongoDB.

MongoDB è un database basato su documenti BSON (binary JSON). Il formato BSON non è altro che il risultato della serializzazione a codifica binaria di un documento JSON. Quest'ultimo è infatti un formato testuale, facilmente leggibile dagli umani ma lento da processare per le macchine. BSON presenta le caratteristiche opposte. Un'altra differenza tra i due formati è la presenta di più tipologie di dato in BSON rispetto a JSON, come ad esempio il DateType [23].

I **documenti** (che concettualmente equivalgono a un record in una tabella relazionale) sono costituiti da coppie Campo-Valore e organizzati in **collezioni**. Essi non devono seguire uno schema predefinito: all'interno di una stessa collezione possono essere presenti documenti con campi diversi. Il valore di un singolo campo, oltre ad assumere i classici Data Type previsti in un database relazionale come *number*, *string* e *date* può anche essere un array di valori, un documento *embedded* o anche un array di documenti. Ciò consente di creare documenti con vari livelli di annidamento ed è particolarmente utile in quei casi in cui si vuole catturare una relazione tra più entità in un singolo documento, senza dover ricorrere al processo di normalizzazione. Un esempio di documento è mostrato in figura 3.3

```
{
    "_id": "5099803df3ff98391",
    "titolo": "Divina Commedia",
    "autore": {                              // Documento Embedded
                "nome": "Dante",
                "cognome":"Alighieri"
    },
    "cantiche": ["Inferno",                  // Array
                "Purgatorio",
                "Paradiso"]
}
```

Figura 3.3: Esempio di documento

MongoDB implementa un linguaggio di query molto espressivo e in grado di effettuare ricerche su tutti i livelli di annidamento di un documento, ma anche su grafi e testi. Una feature importante, specie nel contesto del lavoro presentato, è la possibilità di effettuare query geospaziali. Quest'ultime interessano oggetti **GeoJSON**, ovvero documenti embedded che modellano forme spaziali. Un documento GeoJSON è costituito da un campo *"type"* (Point, LineString, Polygon, MultiPolygon etc) e da un array di coppie di coordinate (*"coordinates"*), riportanti longitudine e latitudine. Una volta definite le forme spaziali è possibile utilizzare operatori geospaziali per comporre le query [24]. Nell'immagine 3.4 è mostrato un esempio di query geospaziale: nella collezione *città* vengono selezionati tutti i documenti il cui campo *location*, modellato come oggetto GeoJSON, risulta "vicino" all'oggetto indicato da *geometry*, ovvero un punto definito da longitudine e latitudine.

```
db.città.find(
    {
      location:
        { $near:
          {
            $geometry: { type: "Point",
                         coordinates: [ -73.9667, 40.78 ] },
            $minDistance: 1000,
            $maxDistance: 5000
          }
        }
    }
)
```

Figura 3.4: Esempio di query geospaziale

Le query geospaziali sono solo una parte dell'espressività che MongoDB consente di ottenere formulando interrogazioni. In questo senso uno strumento molto potente è quello della **Aggregation Pipeline**. Per aggregazione si intende un'operazione che processa valori provenienti da più documenti al fine di elaborare dei risultati. Operazioni tipiche di aggregazione sono la somma, la media, il calcolo del minimo e del massimo [25]. La Pipeline è un'architettura che prevede vari componenti, disposti in sequenza e dunque organizzati in *stage*. Ogni stage lavora sul suo input, costituito da uno o più documenti, ed effettua operazioni quali filtraggio, raggruppamento, lookup. Infine presenta in output il risultato che verrà usato come input dallo stage successivo. Combinando aggregazione e l'architettura a pipeline è possibile svolgere query molto complesse. La tabella 3.1 mostra alcuni degli stage di aggregazione più usati nel nostro progetto.

Altra funzionalità chiave di Mongo è l'utilizzo del **replica set**, ovvero un gruppo di istanze del processo *mongod* operanti sullo stesso dataset. Questo è in accordo ai principi di *data-availability* e *data locality*: avendo a disposizione copie multiple degli stessi dati su diversi server database viene garantita tolleranza agli errori e ai malfunzionamenti. Un replica set è costituito da più nodi che lavorano sugli stessi dati. Di questi, in un preciso momento, solo uno

Stage	Descrizione
addFields	Aggiunge nuovi campi ai documenti
count	Restituisce il numero dei documenti in questo stage
geoNear	Restituisce i documenti ordinati in base alla prossimità a un punto geospaziale
group	Raggruppa i documenti in base a un'espressione
lookup	Svolge un left outer join con i documenti di un'altra collezione
match	Filtra i documenti in base a un'espressione
project	Permette di riformulare la struttura di un documento, aggiungendo e togliendo campi
sort	Ordina i documenti in base a una chiave
unwind	Decostruisce un campo array

Tabella 3.1: Alcuni degli stage che costituiscono la Aggregation Pipeline

ha il ruolo di Primario, cioè riceve tutte le operazioni in scrittura e registra tutti i cambiamenti al suo dataset in log operativi. I nodi secondari replicano le operazioni registrate nell'oplog del nodo primario sui loro dataset, in modo tale da riflettere i cambiamenti avvenuti. Nel caso in cui il nodo primario non fosse disponibile, uno dei secondari ne assumerebbe il ruolo [26].

Poiché i documenti MongoDB sono nel formato BSON e dunque prevedono, come accennato in precedenza, la possibilità di includere oggetti annidati, il rapporto tra entità viene prevalentemente modellato mediante il concetto di de-normalizzazione: invece di distribuire le informazioni in più documenti o collezioni, esse sono tutte presenti nello stesso documento, sotto forma di oggetto embedded. Questa differenza è particolarmente importante nel contesto delle transazioni: intervenire su più entità è un'operazione atomica nel caso di un documento de-normalizzato mentre prevede interventi su più documenti o collezioni se si è scelto di normalizzare la relazione. La struttura di un documento, se ben utilizzata, permette di evitare le transazioni multi-documento nella maggior parte dei casi. Tuttavia MongoDB mette a disposizione, a partire dalla versione 4.0, una API per le transazioni multi-documento che garantisce le proprietà ACID. Come già detto in precedenza, le transazioni ACID richiedono un grande overhead in termini di performance. Considerando che MongoDB è un database distribuito è necessario aggiornare tutte le copie dei documenti che hanno subito modifiche nei vari cluster [27].

3.3 Ambiente di sviluppo frontend

L'architettura di un'applicazione web è naturalmente cambiata del corso degli anni, evolvendosi di pari passo con la complessità delle tecnologie che ne permettono l'implementazione. Un tempo le pagine HTML erano statiche, ovvero non prevedevano un'interazione con l'utente che andasse al di là della semplice visualizzazione del documento. Queste pagine erano dunque modificabili esclusivamente dall'amministratore del sistema o in generale da chi gestiva il web

server. Nel corso del tempo le pagine sono diventate dinamiche: al documento HTML è di solito associato, oltre al foglio di stile, uno o più script. Uno script è un programma nel senso informatico del termine, ovvero scritto mediante un linguaggio di programmazione e non di markup, come HTML (Hypertext Markup Language). La presenza di uno script permette all'utente di interagire dinamicamente con la pagina, che dunque cambierà a seconda delle operazioni svolte dall'utente ed eventualmente processate dal server dell'applicazione. Come conseguenza, le web app sono divenute enormemente più complesse e versatili. Per facilitarne lo sviluppo esse sono state suddivise in due parti, o lati, detti frontend e backend [28].

Il **front-end** è la porzione lato utente, dunque tutto ciò che il web browser visualizza, esegue e in generale gestisce. Il **back-end**, viceversa, è il lato server e svolge tutti i compiti amministrativi, implementa protocolli di sicurezza, gestisce il database, risponde opportunamente alle richieste generate dall'utente. Nel seguente capitolo verranno brevemente descritte alcune soluzioni moderne per lo sviluppo del lato frontend, mostrandone pregi e difetti nel contesto del lavoro presentato in questa tesi. Va subito detto che non tutte le tecnologie presentate solo sullo stesso livello: alcune sono veri e propri framework complessi e ricchi di funzionalità, altre sono semplici librerie che si occupano prevalentemente dell'interfaccia utente (*user interface*, in inglese). Tuttavia esse sono tutte basate su JavaScript, di gran lunga il linguaggio più utilizzato lato front-end.

Consultando varie fonti, come ad esempio StateOfJavaScript [29], è facile capire quali siano le tecnologie basate su JavaScript (o suoi derivati) oggi più voga: React, Angular e Vue.js. Non è presente **JQuery**, la soluzione scelta per questo lavoro, e la motivazione è piuttosto semplice: JQuery è una libreria nata nel 2006, mentre le altre tecnologie proposte sono molto più recenti. Essendo più moderne è naturale pensare di adottarle quando si intraprende lo sviluppo di un'applicazione *ex-novo*. Tuttavia JQuery è risultata, secondo [30], la libreria JavaScript più utilizzata, essendo presente nel 74,4% dei primi 10 milioni di siti più visitati. Questo successo può essere in parte spiegato dal fatto che le web app costruite su si essa non hanno sempre la possibilità o la necessità di passare a un framework più moderno, soprattutto se continuano a svolgere il loro lavoro in modo soddisfacente. Tuttavia non sarebbe giusto ricondurre la massiccia presenza di JQuery nei sistemi web moderni esclusivamente al suo status di tecnologia legacy. JQuery è infatti molto semplice da imparare, soprattutto se paragonata a framework come Angular, è ancora aggiornata, ben documentata e integrata con moltissimi plugin. A livello funzionale, essa semplifica navigazione e manipolazione del **DOM** (Document Object Model), interfaccia utilizzata per rappresentare un documento HTML sotto forma di albero, i cui nodi sono gli elementi costitutivi del documento stesso; consente una facile gestione degli eventi (i.e. il click del mouse su di un bottone); permette di accedere agli elementi HTML ed eventualmente cambiarne gli attributi tramite selettori che conservano la sintassi del CSS; semplifica l'utilizzo delle chiamate asincrone (AJAX); supporta animazioni sia bidimensionali che tridimensionali.

Presentata JQuery e accennati i motivi della sua scelta, che verranno ulteriormente approfonditi a fine capitolo, vengono ora presentate delle possibili alternative.

React (noto anche come React.js) è una libreria JavaScript open source creata da Facebook nel 2013. Viene principalmente utilizzata per la realizzazione di interfacce grafiche nel con-

testo di S.P.A. (Single Page Application) e nel mondo mobile tramite React Native. Poiché si occupa solamente del rendering degli elementi che costituiscono il DOM, lo sviluppo di un'applicazione React richiede l'utilizzo di librerie di supporto, come Redux per lo state management e React Router per il routing [31].

React ha una natura dichiarativa, ovvero permette di scrivere codice che si occupa di descrivere ciò che si vuole ottenere e non come ottenerlo, a differenza del paradigma imperativo. È basata sull'utilizzo dei cosiddetti **components**. Un component è un frammento di UI (User Interface, interfaccia utente), indipendente dagli altri, che accetta in input dei valori e specifica/dichiara un oggetto da renderizzare. A livello di codice, dunque, non è altro che una funzione, o in alcuni casi una classe, scritta in JavaScript che riceve dei parametri (in questo contesto chiamati *props*, da properties) e restituisce un elemento grafico, come ad esempio un ¡div¿. Ordinando logicamente i components e gestendo il loro ciclo di vita mediante i cosiddetti *hooks* (funzioni che si agganciano allo stato di un componente), React consente la creazione di applicazioni modulari e performanti. I component sono infatti riutilizzabili per costruzione e la libreria ne presenta di già predefiniti e testati.

La rapidità di React è legata a una delle sue caratteristiche fondamentali, l'utilizzo di un Virtual DOM. Come già descritto in precedenza Il Document Object Model è un'interfaccia utilizzata per visualizzare la pagina HTML/XML come un albero i cui nodi sono parti del documento stesso. Il programmatore utilizza questa struttura dati come astrazione per lavorare sulle interfacce grafiche che sviluppa. React implementa e gestisce un Virtual DOM, ovvero una struttura dati immagazzinata nella memoria cache che, durante il ciclo di vita dei components, viene continuamente confrontata con il DOM vero e proprio. Se sono presenti differenze tra Virtual DOM e DOM, verranno aggiornati sono gli elementi di quest'ultimo che hanno subito modifiche. Altra caratteristica propria di React è l'utilizzo di JSX, un'estensione della sintassi di JavaScript che permette di scrivere codice simile a quello di HTML e XML.

Ricapitolando, i maggiori vantaggi di React sono 1) la facilità di utilizzo, poiché è basato su JavaScript, 2) l'efficienza con cui viene aggiornata e renderizzata la pagina e 3) l'architettura basata su components, indipendenti e riutilizzabili. I difetti hanno prevalentemente a che vedere con la sua età: essendo una tecnologia relativamente nuova è in continua evoluzione; i frequenti aggiornamenti non risultano sempre ben descritti nella documentazione [32].

Vue.js è un framework progressivo per la creazione di interfacce grafiche. Viene definito progressivo poiché permette al programmatore di scegliere quali librerie o moduli includere nel progetto, a seconda della sua complessità [33]. La libreria principale, o *core*, è quella che si occupa del rendering dichiarativo delle view, ma è possibile aggiungerne altre per svolgere compiti quali routing per SPA, data fetching e state management. Questa caratteristica rende Vue particolarmente versatile e scalabile, poiché può essere usato al contempo come semplice libreria per modernizzare una parte di un'applicazione già esistente oppure come un framework completo e complesso adatto allo sviluppo di un progetto ex-novo [34][35].

Vue presenta numerose caratteristiche in comune con React: la natura dichiarativa, l'architettura basata su components riutilizzabili, la presenza di un Virtual DOM, la core library focalizzata sul rendering delle interfacce utente. Una differenza è rappresentata dal modo in cui vengono costruiti i components: Vue utilizza i Template mentre React il già citato formato JSX. Vue è

particolarmente adatto allo sviluppo di piccole applicazioni, mentre React, grazie al supporto di numerose librerie esterne, può essere usato anche in progetti più grandi; Vue segue il pattern MVVM (Model-View Model-Model) mentre React si concentra esclusivamente sulle View [36].

Angular (da non confondere con Angular 2.0) è un framework sviluppato principalmente da Google a partire dal 2016. Viene utilizzato nel contesto delle SPA. Nato come evoluzione di AngularJS, al contrario di quest'ultimo utilizza TypeScript e non JavaScript. TypeScript è di fatto un superset fortemente tipizzato di JavaScript che viene compilato e dunque permette l'individuazione di eventuali errori a compile-time; JavaScript li individua solo a run-time. È un framework molto versatile in quanto cross-platform: permette lo sviluppo di applicazioni web, mobile e desktop. A differenza di React, che si occupa solo della gestione della interfaccia utente e di Vue, scalabile e progressivo, Angular è un framework *feature-packed*, ovvero contiene al suo interno tutti gli strumenti necessari alla creazione di un'applicazione cross-platform molto complessa, senza dover ricorrere a librerie esterne.

Anche Angular implementa sull'architettura component-based ma presenta molti più strumenti rispetto ai framework/librerie già descritti, come Services, Templates, Metadata, Directives, Pipes. Delle quattro tecnologie descritte fino a ora, Angular è di gran lunga la più complessa e potente: è infatti indicato per lo sviluppo di applicazioni molto grandi. Inoltre è l'unico che richiede allo sviluppatore di imparare un nuovo linguaggio di programmazione TypeScript, sebbene sia molto simile a JavaScript [37][38].

JQuery è stata preferita agli altri framework e librerie descritti in questo capitolo per la semplicità del suo utilizzo, la disponibilità di una documentazione piena e aggiornata ma anche perché alcuni di questi framework sono eccessivamente complessi rispetto agli scopi dell'applicazione, soprattutto considerando che la maggior parte delle funzionalità dell'Atlante Digitale si concentra sul backend. Tuttavia è facile notare che JQuery è funzionalmente molto diversa da React, Vue e Angular: essa non presenta un'architettura component-based, non è focalizzata sull'interfaccia utente e non presenta elementi preconfezionati e già testati che possono essere inseriti nel progetto; ha il solo compito di rendere più facilmente gestibili gli elementi di un DOM. È necessario dunque integrare il suo utilizzo con una libreria che si occupi di problemi come adattività, performance, omogeneità stilistica e che offra dei componenti già pronti da utilizzare. La scelta è ricaduta su Bootstrap.

Bootstrap è una libreria HTML, CSS, e JavaScript tra le più popolari al mondo. Essa mette a disposizione dello sviluppatore un insieme di risorse la cui funzione è quella di semplificare la creazione di una pagina HTML, sia sul piano dello stile che su quello delle funzionalità. Infatti, all'atto della creazione di un sito web, se non venissero utilizzate librerie di supporto uno sviluppatore dovrebbe scrivere non solo tutti i fogli di stile, cercando di mantenere una omogeneità stilistica, ma anche occuparsi di definire tutti gli script che regolano il funzionamento degli elementi e l'adattività della pagina sui diversi dispositivi. Una pagina web può infatti essere consultata da una miriade di dispositivi differenti, ciascuno avente una sua risoluzione, una sua densità di pixel e diverse dimensioni fisiche. Se a questo aggiungiamo la necessità di rendere l'applicazione consultabile da tutti i web browser più utilizzati è facile capire che svolgere una mole di lavoro di queste proporzioni porterebbe naturalmente a molti errori e a

lunghissimi tempi di sviluppo. Ecco perché vengono utilizzate librerie quali Bootstrap, che offre strumenti per facilitare la disposizione degli elementi nella pagina (layout della pagina), come la ben nota griglia, fogli di stile già compilati che permettono di implementare un design moderno, componenti dalle funzionalità ben definite e testate, ulteriormente configurabili dal programmatore. Bootstrap dunque si occupa di rendere l'applicazione web adattiva e compatibile su tutti i browser. Anche in questo caso è presente una documentazione ben scritta e una community ampia. Alcuni difetti della libreria sono dovuti proprio alla sua facilità di utilizzo: poiché mette a disposizione template pre-compilati si può notare, navigando sul web, che molti siti hanno un design molto simile. Questo problema può essere superato personalizzando il template. Altro problema è rappresentato dal fatto che la libreria nella sua interezza offre più funzionalità di quelle che vengono utilizzate effettivamente dall'applicazione, costituendo un peso inutile. È tuttavia possibile scegliere solo un sotto-insieme di componenti da importare [39].

L'utilizzo combinato di JQuery e Bootstrap è adatto agli scopi dell'applicazione presentata: sviluppo semplice e rapido, design moderno, utilizzo di strumenti e componenti già testati e affidabili. Gli altri framework sono sì più moderni ma spingono molto sulla dinamicità e reattività della pagina, elementi sicuramente importanti ma che complicano il lavoro si sviluppo se non fondamentali. La tabella 3.2 ricapitola vantaggi e svantaggi degli strumenti presentati in questo capitolo.

Tecnologia	Caratteristiche Chiave	PRO	CONTRO	Indicato per
React	- Architettura a components - Focus su View - Virtual DOM - One Way Data Binding - JSX	- Prestazioni - SEO friendly - Facilità di utilizzo - Molti plugin disponibili	- Sviluppo veloce - Documentazione non sempre aggiornata	-Piattaforme mediamente complesse, community based
Vue.js	- Architettura a components - Uso di Templates - Framework progressivo	- Prestazioni - Facilità di utilizzo - Versatilità - Scalabilità	- Pochi plugin - Community piccola - Eccessiva flessibilità	- Applicazioni semplici
Angular	- Future Packed - Cross Platform - Template flessibili	- Completo	- Complesso, per via di TypeScript - Verboso - Non SEO friendly	- SPA, applicazioni complesse
JQuery + *Bootstrao*	- Gestione elementi DOM, eventi, selettori, richieste AJAX - Componenti pre configurati - Stile moderno	- Semplice organizzazione del layout - Buona documentazione	- JQuery è una libreria attempata	- Applicazioni con focus su funzionalità backend, che non richiedono grande dinamicità e reattività

Tabella 3.2: Confronto tra tecnologie front end

3.4 Ambiente di sviluppo backend

Il frontend riveste un'importanza cruciale relativamente all'esperienza utente, poiché questo ultimo interagisce principalmente con l'interfaccia grafica e dunque si aspetta che questa sia semplice ed intuitiva da utilizzare, reattiva ed adattiva ma anche moderna in termini di design. Il successo di molte applicazioni è sempre più associato alla cura della User Experience. Tuttavia, a livello operativo, è il **backend** il nucleo logico e funzionale di un'applicazione web. Questo perché svolge una serie di compiti eterogenei che vanno dall'interazione con il database alla risposta delle richieste utente, il tutto seguendo e implementando protocolli di sicurezza per prevenire attacchi, organizzando il traffico utente, raccogliendo statistiche di utilizzo e molto altro. Anche lato backend è dunque fondamentale la scelta di un framework: le qualità più

richieste sono velocità di sviluppo, spesso ottenuta automatizzando quelle operazioni che è necessario ripetere più e più volte; la scalabilità, che permette di abbattere i costi; la robustezza, fondamentale per continuare ad erogare un servizio anche in presenza di errori; la sicurezza, specie nei contesti in cui vengono analizzate ed utilizzate informazioni sensibili; flessibilità, per consentire l'integrazione con altri strumenti di utilità.

Prima di decidere quale framework utilizzare è tuttavia necessario pensare a quale architettura software sia più adatta alla tipologia di applicazione che si vuole sviluppare. L'architettura descrive logicamente la struttura e il funzionamento del sistema, definendo l'interazione tra componenti e un insieme di metodi, pratiche e protocolli da seguire. Il ricorso a un pattern architetturale non solo consente di aver già pronto uno schema di base per la costruzione del sistema ma facilita l'interazione tra sviluppatori. Seguire le stesse regole, definite dall'architettura, permette di lavorare più semplicemente su codice scritto da altri. Ulteriori informazioni riguardanti la architettura del sistema verranno date nel Capitolo 4. Prima di descrivere il framework di backend utilizzato è dunque importante presentare quale architettura implementa: nel nostro caso si tratta della **Model-View-Controller**, abbreviata in **MVC**.

Il **Model** è la parte che concerne l'utilizzo e la gestione dei dati, i quali costituiscono lo stato di un'applicazione. È dunque il componente centrale dello schema, poiché definisce la logica e le regole dell'applicazione. Fornisce dei metodi per interrogare e modificare la base di dati, ma non ha alcun rapporto con lo strato di presentazione, ovvero il livello dell'interfaccia utente. È il Controller a richiedere i servizi del Model, a seconda del tipo di richiesta ricevuta dall'utente.

La **View** riguarda la presentazione, o visualizzazione dell'informazione. Più View possono essere associate agli stessi dati, a seconda del contesto. Generalmente le View sono realizzate tramite template, opportunamente compilati e scelti dal Controller, che vengono successivamente renderizzati dal browser come pagine HTML.

Il **Controller** è il mediatore tra utente e sistema. Interagendo con la View, l'utente in realtà invia richieste al Controller, che nel contesto web è un semplice indirizzo sul server, ovvero un URL. Analizzata la richiesta il Controller interagisce con il Model grazie alle funzioni che quest'ultimo espone. L'interazione può prevedere una semplice interrogazione della base di dati o anche un processo più lungo che può portare alla modifica dello stato dell'applicazione. Svolto il lavoro sul Model, il Controller si occupa di rispondere alla richiesta utente selezionando e configurando la View più adatta [40].

La suddivisione di un'applicazione nelle tre parti descritte poco sopra ha lo scopo di renderle indipendenti tra di loro e dunque più semplici da testare, modificare ed eventualmente sostituire. Si pensi ad esempio al caso tipico di un'applicazione web multipiattaforma e dunque consultabile da più device: uno smartphone, un tablet e un pc hanno caratteristiche diverse, in primis per quanto riguarda la dimensione dello schermo, ma anche per quanto concerne l'interazione con l'interfaccia. Sebbene l'applicazione svolga in tutti questi casi le stesse funzioni utilizzando lo stesso data model è necessario prevedere un'interfaccia utente diversa per ciascun dispositivo. Utilizzando il MVC è possibile sviluppare un sistema che abbia un solo data model e un Controller che si occupi di presentare all'utente la View più adatta alle sue esigenze. Nell'immagine 3.5 è sintetizzata l'interazione tra i componenti.

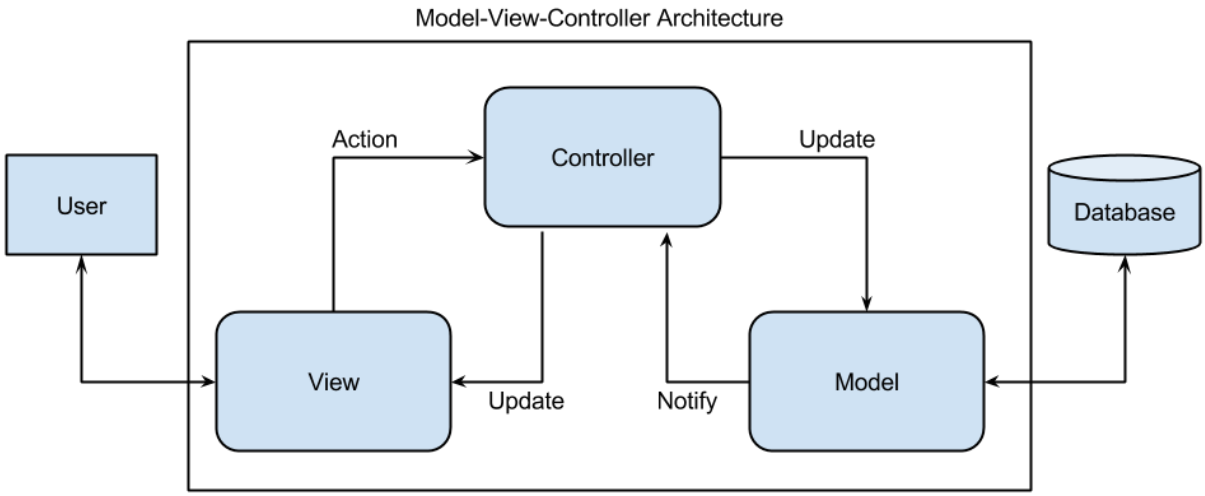

Figura 3.5: Il pattern Model View Controller (immagine dal web)

Poiché il MVC si è facilmente imposto come soluzione ideale nello sviluppo di una applicazione web, sono nati framework costruiti su di esso. Alcuni esempio sono Django, Laravel, Ruby on Rails, CakePHP, Phoenix. Di questi, la parte restante del paragrafo ne descriverà due: Ruby on Rails e quello effettivamente utilizzato per lo sviluppo dell'Atlante Digitale, Django.

Ruby on Rails, spesso abbreviato in Rails, è un framework open source scritto in Ruby.

Ruby è un linguaggio interpretato creato allo scopo di trovare un equilibrio tra paradigma imperativo e quello funzionale. Nonostante sia stato creato nel 1995 ha conosciuto il successo solo recentemente, appunto grazie a Rails, e oggi figura tra i dieci linguaggi maggiormente utilizzati. Sue caratteristiche principali sono 1) il paradigma a oggetti puro, ovvero ogni componente del linguaggio, sia esso una classe o una costante, è un oggetto e dunque possiede metodi e 2) il dinamismo degli oggetti, il cui tipo non è definito dalla classe che lo ha istanziato (come avviene in linguaggi come C++) ma dai metodi che esso possiede, metodi che possono essere aggiunti o modificati anche a run-time [41].

Tornando al framework in sé, i principi cardine alla base della sua creazione sono "don't repeat yourself" ("non ripeterti") e "Convention over configuration" ("la convenzione è da preferire alla configurazione"). Il primo, auto-esplicativo, impone di evitare di riscrivere porzioni di codice. Essendo un framework full-stack, realizzato quindi per poter sviluppare applicazioni database-driven basate sul pattern MVC, tutte le componenti del sistema sono integrate e collegate tra di loro automaticamente, senza richiedere l'intervento del programmatore. "Convention over configuration" sta a significare che la configurazione è richiesta solo per quelle parti che deviano da un certo standard. Ai file di configurazione sono dunque preferiti l'uso di convenzioni, la proprietà di riflessione del codice (ovvero la capacità di descrivere sé stesso e la sua struttura) e ambienti di esecuzioni dinamici. Altri punti di forza del framework sono la possibilità di scrivere facilmente meta-programmi in Ruby, ovvero programmi capaci di utilizzare altri programmi (inclusi sé stessi) come dati su cui lavorare; la presenza di strumenti per l'automatizzazione dei test; lo scaffolding (traducibile come impalcatura), ovvero la possibilità, specie nelle prime fasi dello sviluppo, di scrivere del codice temporaneo con il solo fine di poter eseguire subito quelli che sono i componenti più importanti dell'applicazione; la

licenza open-source che in ogni ambito si accompagna alla presenza di una estesa community e alla disponibilità di ricche librerie con cui estendere le funzionalità del sistema; la velocità di apprendimento e, contemporaneamente, quella di sviluppo; l'assenza di una fase di compilazione [42].

Per quanto riguarda i difetti, Rails eredita quelli di Ruby, ovvero scarse performance, tipiche di un linguaggio interpretato e il mancato utilizzo di thread nativi, che richiedono una virtual machine sulla quale essere eseguiti.

Come già accennato il framework utilizzato per lo svolgimento di questo progetto non è Rails ma Django. Il perché di questa preferenza verrà riportato nella parte conclusiva del capitolo, dopo aver fornito una introduzione, sufficientemente dettagliata, di Django.

Django è un web framework basato su Python che incoraggia il rapido sviluppo di applicazioni dal design pulito e pragmatico. Mette infatti a disposizione strumenti di varia natura che consentono di evitare di scrivere codice per componenti dal funzionamento standard ("reinventare la ruota") e di facilitare l'implementazione di alcune caratteristiche sempre richieste da applicazioni di una certa complessità. Un esempio è la presenza di funzioni di sicurezza built-in, che aiutano ad evitare attacchi di Clickjacking, Cross-Site Scripting, SQL Injection e Cross Site Request Forgery; o anche il supporto alla internazionalizzazione, ovvero l'insieme delle procedure da effettuare per rendere l'applicazione consultabile in diverse lingue, in diverse parti del mondo [43]. Prima di passare a una presentazione più dettagliata degli strumenti del framework in sé è necessario descrivere il linguaggio su cui è basato e da cui eredita, almeno in parte, vantaggi e svantaggi.

Python è uno dei linguaggi di programmazione più utilizzati al mondo e deve il suo successo a una molteplicità di fattori, su cui spiccano la semplicità di utilizzo, la versatilità e la disponibilità di potentissime librerie attinenti ai più svariati ambiti. È un linguaggio molto facile da imparare poiché supporta la non-tipizzazione delle variabili, la gestione automatica della memoria tramite un *garbage collector* e una sintassi estremamente asciutta e leggibile, al punto da sembrare pseudo-codice. Per un programmatore alle prime armi, infatti, il sistema dei tipi, la gestione della memoria e le regole sintattiche costituiscono i primi scogli nell'apprendimento. Contiene concetti provenienti da numerosi paradigmi di programmazione: quello orientato agli oggetti, il funzionale, l'imperativo, lo strutturato. Viene di solito inserito nel novero dei linguaggi interpretati sebbene il codice sorgente passi prima attraverso una fase di pre-compilazione. Il bytecode così prodotto viene poi utilizzato dopo la prima esecuzione del programma per evitare che il sorgente venga re-interpretato ogni volta. La sua natura multi-paradigma lo rende un linguaggio "universale", cioè adottabile in tutti i contesti di sviluppo: web, desktop, mobile, gaming e soprattutto machine learning e data analysis [44].

Ovviamente un linguaggio così espressivo e a un livello di astrazione così spinto presenta dei limiti in termini di performance. In contesti in cui la velocità di esecuzione è di primaria importanza è meglio affidarsi a linguaggi quali C++, comunque di alto livello ma che seguono la filosofia della "zero cost abstraction": tutti i costrutti del linguaggio non devono, almeno come comportamento di default, portare a una diminuzione delle prestazioni.

Django è stato sviluppato nel 2005 con l'obiettivo di semplificare la creazione di portali di notizie. Oggi è il framework utilizzato da giganti quali Instagram, Pinterest, Youtube,

Dropbox, Spotify e molti altri. Tuttavia presenta ancora oggi delle funzionalità legate al suo passato: il framework genera automaticamente un **pannello amministrativo** alla creazione dell'applicazione. Grazie ad esso è possibile creare, modificare ed eliminare tutte le entità e gli oggetti del sito web, riportare le operazioni in un file di log e svolgere tutti i compiti riguardanti l'autenticazione degli utenti. Questi includono la creazione di account, la gestione dei cookie e delle sessioni e l'organizzazione in gruppi con diversi livelli di privilegio, per menzionarne alcuni. Il pannello amministrativo è naturalmente legato al sistema di sicurezza cui si è accennato poco sopra, che garantisce protezione da numerose tipologie di attacchi informatici.

Essendo un framework full-stack, è naturalmente costruito per lavorare con un DBMS. Originariamente era supportato solamente PostgresSQL, ma oggi la compatibilità è estesa non solo ad altri DBMS relazionali, come MySQL, SQLite, Microsoft SQL Server, Oracle ma anche a quelli non relazionali, come MongoDB. L'integrazione con il database è realizzata mediante la tecnica dell'Object-Relational Mapping (ORM), che consiste nel fornire un'interfaccia, orientata agli oggetti, che offre servizi relativi alla persistenza dei dati. Detto in altri termini, l'ORM prevede la creazione dello schema del database a partire da classi, scritte in un linguaggio di programmazione, che ne descrivono il data model. In questo modo è possibile lavorare a un livello di astrazione che copre le varie differenze tra i diversi DBMS usati [45]. Sebbene il concetto di ORM sia legato al mondo dei database relazionali, esistono strumenti, come MongoEngine, che presentano funzionalità simili e riadattate al contesto NoSQL. L'ORM di Django è uno dei suoi aspetti al contempo più apprezzati e criticati: esso semplifica la scrittura delle entità di una base di dati ma è onnipresente e non può essere disabilitato.

Altra componente importante di Django è il **REST framework**, un toolkit utilizzato per creare REST API. Una REST API è un'interfaccia conforme all'architettura REST (Representational State transfer), costruita sui seguenti principi [46]:

1. Uniformità delle richieste: ogni risorsa del server deve essere identificata da un singolo URI (Uniform Resource Identifier), a prescindere dalla provenienza delle richieste.

2. Disaccoppiamento tra client e server: il client deve conoscere solo l'URI della risorsa, il server deve solo fornire dati al client via HTTP.

3. Mancanza di stato: una richiesta deve contenere tutte le informazioni relative al suo processamento; il server non deve immagazzinare dati relativi alle richieste client.

4. Uso della cache: se possibile, le risorse devono essere riposte nella cache sia lato client sia lato server.

5. Architettura a strati: tra richiesta client e risposta server devono essere presenti degli intermediari.

Grazie al DRF (Django Rest Framework) lo sviluppatore ha accesso a un gran numero di API, ciascuna presentante una buona documentazione e sostenuta da una nutrita community. Altro aspetto da approfondine è la modalità con la quale Django implementa il pattern MVC.

Quando un utente richiede una risorsa (che può essere una pagina, un file o altro), essa viene identificata da un indirizzo. *L'URL Dispatcher*, opportunamente configurato nel file chiamato *urls.py*, si occupa di associare l'indirizzo richiesto alla View preposta al suo processamento. La View processa dunque la richiesta dell'utente: se è abilitato il sistema di caching, essa fornisce direttamente la risorsa, se già disponibile. Se non lo è interagisce con il Model tramite i metodi che questo espone e, terminate le operazioni, risponde all'utente. La risposta può essere una pagina; in questo caso la View si occupa di compilare opportunamente un Template, che verrà restituito all'utente e renderizzato dal browser. Questa descrizione non sembra coincidere perfettamente con il pattern MVC, visto che la View svolge parte del lavoro riservato al Controller, compresa l'interazione con il Model. E in effetti, questo pattern prende il nome di **Model-View-Template**. Un Template non è altro che un modello di pagina HTML da compilare opportunamente nel momento in cui viene richiesto. Il suo utilizzo rende più semplice il lavoro di sviluppo front-end. Nel contesto MVT è dunque il Template ad assumere il ruolo che la View ha nel MVC, ovvero quello di strato di presentazione.

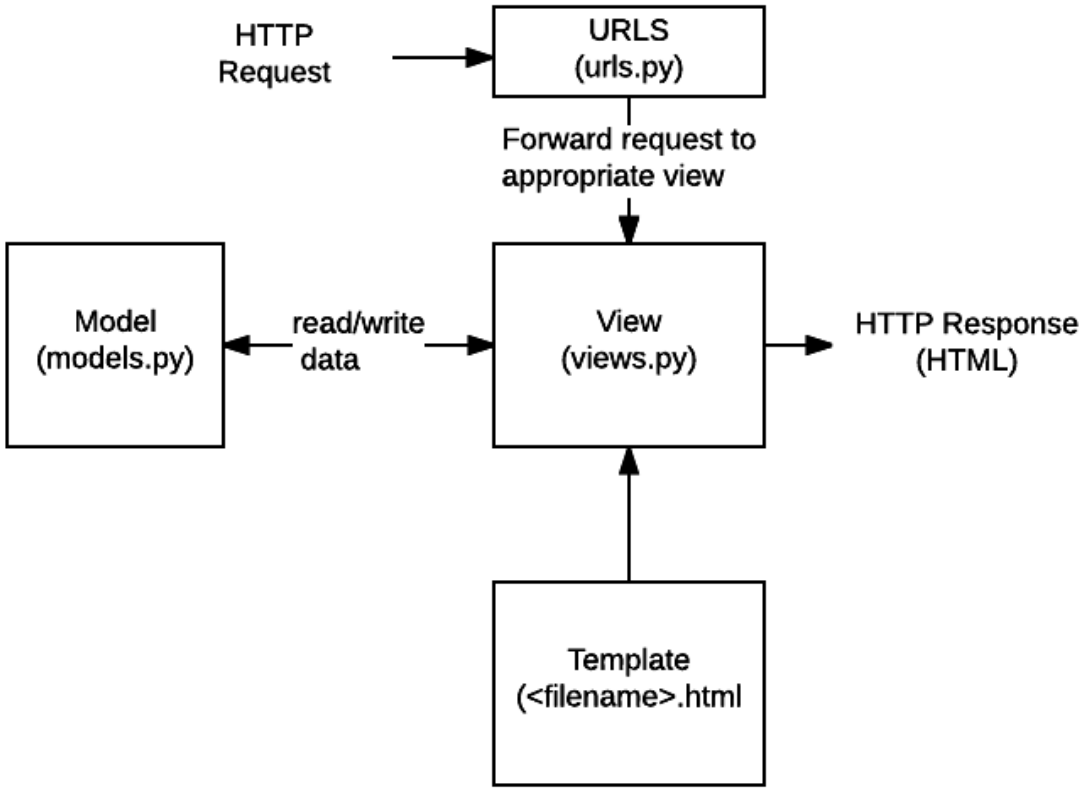

Figura 3.6: Pattern Model-View-Template di Django (immagine dal web)

Per terminare la discussione su Django, è opportuno presentarne qualche difetto. Django è un framework monolitico, dunque richiede memoria, spazio e CPU; è inadatto a progetti molto piccoli, in quanto adotta la filosofia "batterie incluse", ovvero si propone di sviluppare un'applicazione in tutta la sua interezza, non permettendo di concentrarsi solo sugli aspetti definiti "core"; pecca nell'uso di convenzioni (principio fondante di Rails), dunque richiede al programmatore di configurare manualmente alcune componenti del sistema. Ciononostante, è stato preferito a Ruby on Rails, con il quale condivide comunque numerose caratteristiche,

perché più adatto allo sviluppo di progetti grandi ed estendibili; per l'uso del Model-View-Template, più semplice del MVC soprattutto per quanto riguarda le funzionalità del Controller; per la semplicità di scrivere codice in Python, linguaggio usato in molti più contesti rispetto a Ruby; infine per la presenza, già discussa, di strumenti quali il pannello amministrativo, il sistema di autenticazione e sicurezza, il toolkit per la realizzazione di REST API.

3.5 Conclusioni

In questo capitolo sono state presentate tutte le tecnologie alla base della realizzazione dell'Atlante. La tabella **??** ricapitola alcuni dei motivi che hanno portato alla scelta delle soluzioni adottate.

	Soluzione scelta	**Motivazioni della scelta**
Piattaforma di deployment	Docker	I container sono più leggeri, maneggevoli e semplici da utilizzare rispetto alle macchine virtuali.
Database	MongoDB	Un database NoSQL è il più indicato per trattare dati dalla strutturaeterogenea e dinamica come quelli dell'Atlante Digitale. È possibile svolgere query geospaziali e sono supportate le transazioni ACID multi-documento.
Ambiente frontend	JQuery + Bootstrap	Le funzionalità dell'Atlante sono concentrate sul backend, dunque non vi è necessità di usare framework e librerie complesse. JQuery e Bootrstrap garantiscono semplicità e design moderno.
Ambiente backend	Django	Python è un linguaggio semplice da imparare. Pannello amministrativo, preato automaticamente, toolkit per la creazione di API REST, Model View Template, protocolli di sicurezza già integrati che evitano alcune tipologie di attacchi informatici.

Tabella 3.3: Tabella di riepilogo delle tecnologie scelte

Capitolo 4

Architettura dell'Atlante Digitale

Il concetto di architettura di un sistema software è stato già presentato nel Capitolo 3, nel contesto della scelta del framework per il lato backend dell'applicazione. Tuttavia il MVC non è il solo pattern architetturale implementato: un sistema informativo moderno possiede una complessità tale da poter essere analizzato secondo numerosi punti di osservazione. Ciascuno di essi permette di concentrarsi su un aspetto diverso del sistema. Il MVC e MTV sono pattern che fanno riferimento solamente al rapporto tra le tre componenti logiche del sistema nel riguardanti l'interazione tra frontend e backend. Non descrive, però, come vengono strutturati i servizi dell'applicazione, né come viene composta e gestita l'interfaccia utente. Del resto, nel contesto della costruzione di un edificio, da cui è mutuata la parola architettura, vengono prese decisioni riguardanti non solo la struttura, ma anche il design, le funzionalità, l'integrazione con l'ambiente circostante. La stessa cosa accade nell'ambito dello sviluppo di sistema informativo moderno, che dunque verrà costruito seguendo non un solo pattern, ma molteplici. Oltre al MTV, infatti, il sistema costruito implementa le architetture client-server, a microservizi e component-based. Se le ultime due sono state già descritte rispettivamente nell'ambito della piattaforma di deployment e dell'interfaccia utente, è bene presentare brevemente la prima.

Le applicazioni web sono costruite sull'architettura **client-server**: il client è l'entità che dà inizio alla comunicazione; il server si occupa di rispondere. Client e server sono concetti astratti; essi non corrispondono necessariamente a macchine fisiche ma sono spesso definiti utilizzando la parola servizio: un singolo server fisico può avere in esecuzione più processi, ciascuno con il compito di server. Sebbene questo pattern non sia legato a una tecnologia specifica, è naturale associarlo alla rete Internet, l'infrastruttura oggi più utilizzata in ambito comunicativo.

Il client si interfaccia al sistema, e viceversa, mediante l'uso di un browser web, che si occupa di generare le richieste a fronte di azioni compiute dall'utente, ma anche interpretare le risposte del server, ad esempio renderizzando le le pagine HTML. L'applicazione web vera e propria viene eseguita in isolamento da un server web, servizio software che ha l'incarico di ricevere le richieste dei vari client, organizzarle in una coda (o altra struttura dati), processarle seguendo un algoritmo che ne determina la priorità e generare una risposta. Quest'ultima può assumere le forme di una pagina HTML, magari risultato della compilazione di un Template, ma anche essere un file o un documento con un formato standard, come JSON o XML. In questo ambito

si inserisce il pattern MTV, dunque in un contesto più specifico rispetto all'interazione generale tra utente e sistema.

Il ricorso ai pattern architetturali è fondamentale per costruire un sistema complesso. Seguire un modello permette lo sviluppo di software non solo funzionale ma anche prestante, flessibile, integrabile, riciclabile ed economico. Generalmente un Sistema Informativo viene suddiviso in più parti, legate tra di esse, ciascuna modellabile seguendo un certo numero di pattern architetturali. Una divisione significativa è quella che prevede la presenza di Hardware, Software, Base di Dati, Rete e Risorse Umane. Oggetto di questo capitolo sono quelle relative a Software e Database.

Come già fatto nel Capitolo 3, iniziamo dalla fase di Deployment. Il progetto è basato su più container, mostrati in figura 4.1. Quelli indicati con il nome Mongo e seguiti da un numero rappresentano il **replica set**, di cui si è già discusso in precedenza. **HBweb** è il container relativo a Django. Esso contiene dunque tutti i file, le librerie e i plugin di cui necessita l'applicazione web. È il container principale per quanto riguarda il lavoro dello sviluppatore: il codice scritto, sia frontend che backend, è contenuto in esso. **Nginx** è un server HTTP che svolge anche le funzioni di reverse proxy e mail proxy. Viene utilizzato per la gestione di risorse statiche e file indice. Oltre ad essi è presente un ulteriore container, **hbmongosetup**, che ha la sola funzione di configurare inizialmente il replica set.

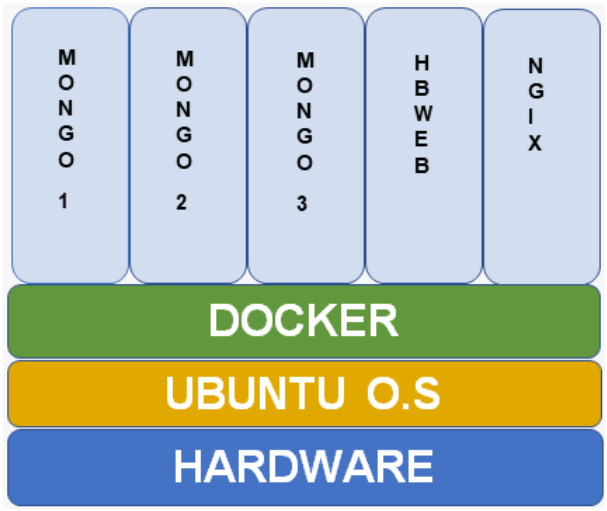

Figura 4.1: Container utilizzati dall'applicazione

I container sono gestiti mediante il tool Compose. Questo è basato su un file di configurazione nel formato YAML, di cui è mostrato un esempio in figura 4.2. Il file risulta leggibile anche a chi non ha esperienza con lo strumento: il container dal nome Ngnix viene costruito a partire dai file presenti nella cartella indicata in *build*; per la persistenza dei dati usa i volumi indicati da *volumes*, a loro volta definiti più in giù nello stesso documento YAML; essendo un server HTTP utilizza le porte 80 e 443, per la versione HTTPS. Sono poi presenti variabili di ambiente, il nome del cluster cui appartiene e le dipendenze rispetto agli altri container.

```
nginx:
  build: ./nginx-pagespeed/
  restart: always
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/app/nginx.conf
    - ./nginx/conf:/etc/nginx/app/conf_app/
    - django_admin_static:/django_admin_static
    - ./web/histbuilding/static:/static
    - ./nginx/certificate:/ssl
    - tinymce:/tinymce
    - django_tinymce:/django_tinymce
    - hbassets:/assets
  ports:
    - "80:80"
    - "443:443"
  depends_on:
    - hbweb
  networks:
    - hb_cluster
```

Figura 4.2: Sezione di file YAML relativa al container Ngnix

4.1 Sistema Database

Nelle prime fasi di sviluppo di un sistema software, dopo la raccolta di requisiti e funzionalità e prima di decidere quali tecnologie utilizzare, è necessario costruire un modello dei dati o data model.

Un data model è un modello astratto utilizzato per descrivere tutti gli aspetti dei dati che il sistema si ritroverà a dover maneggiare, dalla struttura alle loro relazioni reciproche. Definire correttamente il data model è fondamentale per evitare alcuni problemi che non è facile prevedere nei primi stadi di sviluppo di un sistema informativo ma che emergono più tardi, quando è molto più complicato gestirli. In particolare, le politiche (o *business rules*) di un sistema, che rientrano nella logica applicativa, sono strettamente indipendenti dal data model. Modificarle comporta la ridefinizione delle componenti del sistema che si interfacciano con la base di dati; lavoro non semplice, poiché tipicamente riguarda molte linee di codice. Inoltre, se non vengono adottati dei protocolli o standard nel design di un data model, è possibile che il sistema si ritrovi a gestire dati con formati e definizioni particolari, che non sono facilmente condivisibili tra i vari componenti dell'applicazione. Altro problema generato da un data model mal compilato è il rischio di ritrovarsi a gestire entità e funzioni duplicate o molto simili tra di loro, che di conseguenza causano ridondanza [47].

A seconda del livello di astrazione è possibile considerare tre diverse tipologie di data-model:

1. Data model concettuale: è il livello più alto, strettamente deputato alla creazione di una semantica del dominio. Indipendente dal DBMS, esso definisce concetti e regole del sistema, senza però accennare alla natura implementativa.

2. Data model logico: è qui che vengono modellate le entità e le loro relazioni, appoggiandosi ai concetti di tabelle, classi orientate agli oggetti, tag XML o altro. Anche a questo livello c'è indipendenza dal DBMS, anche se è già chiara la natura relazionale o meno dei dati.

3. Data model fisico: fornisce tutti i dettagli implementativi, strettamente legati alla scelta del DBMS.

Nella figura 4.3 è mostrato il data model logico del sistema. La rappresentazione a grafo, in cui le entità costituiscono i nodi e gli archi le relazioni tra di esse, è particolarmente adatto alla descrizione di dati eterogenei. Nodo centrale è il Palazzo, necessariamente legato a una Località e caratterizzato dagli Elementi Descrittivi. Da notare come più Elementi Descrittivi possano essere collegati alla stessa Fonte. Questa relazione, unità alla possibilità di configurare liberamente gli Elementi Descrittivi, costituisce una delle peculiarità maggiori del sistema.

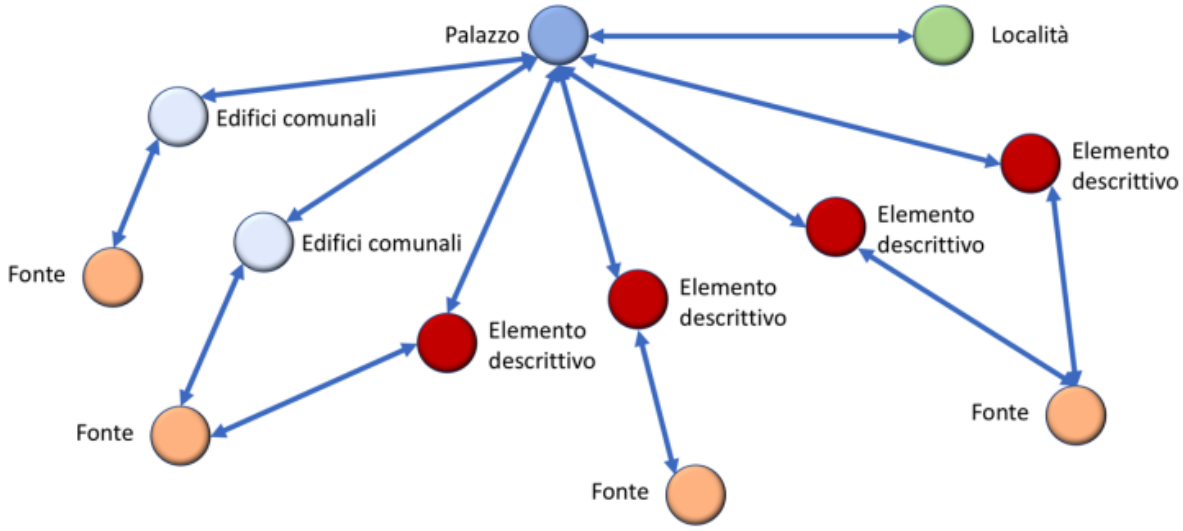

Figura 4.3: Data Model logico del sistema

Poiché il DBMS scelto è MongoDB, le entità sono organizzate in collezioni di documenti che non seguono uno schema predefinito. Tuttavia i documenti appartenenti a una data collezione possono contenere o meno un certo numeri di campi già stabiliti, su cui l'utente non può intervenire. Dunque nelle seguenti righe la parola schema assumerà il significato di insieme di campi per i quali un documento può avere o meno un valore. Discorso a parte meritano gli Elementi Descrittivi, la cui struttura è liberamente configurabile dall'utente. La differenza sarà più chiara dopo aver presentato le caratteristiche specifiche di ogni collezione.

Il **Palazzo** (schema in tabella 4.1) è il principale oggetto di studio dell'Atlante. È l'unica entità avente due campi identificativi, *ID* (1) e *Slug* (14). Quest'ultimo è realizzato come una stringa composta dal nome del palazzo (3) e un numero progressivo, in modo tale da permette di discernere Palazzi aventi lo stesso nome. In una stessa Località non possono esistere più Palazzi con lo stesso nome, ma è ovviamente possibile che esistano palazzi con lo stesso nome in Località diverse. Lo Slug costituisce l'ultima parte dell'URL della pagina web ed è fondamentale che sia leggibile soprattutto per i motori di ricerca. I web crawler, infatti, usano l'URL per estrapolare il contenuto della pagina HTML con il fine di indicizzarla e posizionarla nel web. Il campo *Riconoscibilità* (2), un booleano nella versione precedente dell'applicazione, è una stringa poiché è stato introdotto il concetto di riconoscibilità parziale. Questo campo determina o meno la presenza dei campi [8-10], che non avrebbe senso specificare se il Palazzo non fosse almeno parzialmente riconoscibile. *Geolocalizzazione* (5) è un oggetto embedded costituito da due campi, Tipo e Coordinate. Il primo indica la forma del Palazzo, che può essere un punto o un poligono; il secondo è una lista di coppie di coordinate, indicanti longitudine e latitudine. Il campo *Recensione* (6) costituisce un esempio di de-normalizzazione poiché è l'insieme di tutte le recensioni, nelle varie lingue, del Palazzo. Invece di dedicare un'intera collezione a quest'entità, essa è divenuta un campo all'interno di Palazzo, con il fine di modellare la relazione in un unico documento. Ogni recensione è costituita dai campi Lingua e Testo. *ID Località* (15), al contrario, è un esempio di normalizzazione, poiché costituisce la chiave esterna (FOREIGN KEY) che lega il Palazzo alla Località cui appartiene. In questo caso si è scelto di non seguire uno dei principi dei DBMS NoSQL, che prescrive la duplicazione dei dati per aumentare le performance, poiché il sistema non si trova a gestire enormi volumi di dati e dunque non rischia di subire bruschi cali di prestazione.

	Palazzo
1	ID: stringa alfanumerica
2	Riconoscibilità: stringa
3	Denominazione moderna: stringa
4	Denominazioni antiche: lista di stringhe
5	Geolocalizzazione: oggetto embedded
6	Recensione: lista di oggetti embedded
7	Descrizione di sintesi: stringa
8	Destinazione d'uso: oggetto embedded
9	Criteri e strumenti di datazione: oggetto embedded
10	Stato attuale di conservazione: oggetto embedded
11	Processo formativo rispetto al contesto: stringa
12	Rapporto luogo di edificazione con struttura insediativa antica: stringa
13	Rapporto con aree funzionali coeve: stringa
14	Slug: stringa
15	ID Località: stringa alfanumerica

Tabella 4.1: Schema Palazzo

La **Località** (schema in tabella 4.2) indica dove sono presenti i Palazzi o gli Edifici Comunali. Il campo principale nei casi d'uso che riguardano questa entità è *Geolocalizzazione* (5). È strutturalmente uguale a quello presente nella tabella 4.1 relativa ai Palazzi ma, a differenza di quest'ultimo, viene utilizzato per svolgere query GeoSpaziali. È dunque possibile chiedere quanti e quali Palazzi o Edifici Comunali sono presenti nel poligono che definisce i contorni della Località o anche quali sono solamente vicini ad essa. Anche per la Località è prevista la possibilità di definirla come punto e non come poligono. Questa scelta deriva dal fatto che non sempre è possibile stabilire con certezza i contorni di una Località che magari non esiste più e di cui abbiamo informazioni provenienti solo da testi antichi. È però lecito chiedersi cosa succede quando si vuole associare un Palazzo o un Edificio Comunale a una Località definita da un punto. Poiché queste entità sono legate alla Località mediante chiave esterna, non verrebbe effettuata una query GeoSpaziale ma utilizzata una pipeline di aggregazione con funzione di Inner Join. Il ricorso alla pipeline però permette anche di porre Palazzi ed Edifici al di fuori di Località ben definite da un contorno. Anche questo è tuttavia un requisito funzionale, poiché è sensato, nel contesto degli edifici storici, poter dire che un convento, ad esempio, si trova in una data Località pur essendo fuori dalle sue mura.

	Località
1	ID: stringa alfanumerica
2	Denominazione attuale: stringa
3	Denominazioni precedenti: lista di stringhe
4	Prima attestazione dell'organismo comunale: oggetto embedded
5	Geolocalizzazione: oggetto embedded

Tabella 4.2: Schema Località

La collezione **Edificio Comunale** (schema in tabella 4.3) è presente per raggruppare quegli edifici utili a caratterizzare il contesto di una Località ma che non hanno la stessa importanza dei Palazzi. Al contrario di questi, infatti, gli Edifici Comunali non hanno una scheda dedicata, non presentano alcuna sintesi e non sono visualizzabili sulla mappa principale dell'applicazione.

	Edificio Comunale
1	ID: stringa alfanumerica
2	Denominazione attuale: stringa
3	Funzione: stringa
4	Identificazione e denominazione: lista di oggetti embedded
5	Altro: stringa
6	ID Località: stringa alfanumerica

Tabella 4.3: Schema Edificio Comunale

Le **Fonti** sono un elemento fondamentale dell'Atlante, come è possibile notare dal Data Model 4.3. Esse costituiscono il materiale di ricerca contenente i riferimenti agli Elementi Descrittivi. L'applicazione prevede sette diverse tipologie di fonti configurate staticamente, frutto

di un lavoro di comunicazione tra sviluppatori e architetti, end-user del sistema. Delle sette la più importante è quella delle Fonti Bibliografiche: su di essa sono mappate tutte le fonti importate da Zotero, software per la ricerca e gestione di materiale di studio. È comunque possibile inserire manualmente una Fonte Bibliografica senza passare per il tool di importazione. Nella tabella **??** è mostrato lo schema della tipologia *Letteratura Grigia*, che presenta meno campi rispetto alle altre ma è comunque rappresentativa della collezione. Il campo *Autori* (2) racchiude una lista di oggetti, ciascuno avente i campi Nome e Cognome. Stesso discorso vale per *Data*, che però è un oggetto unico con attributi Tipologia (Data Completa, Mese e Anno, solo Anno) e Valore (la data in sé). Il campo *Path Documento* (5) indica il percorso nel file system del server dell'eventuale documento associato alla fonte.

	Fonte (Letteratura Grigia)
1	ID: stringa alfanumerica
2	Autori: lista di oggetti embedded
3	Titolo: stringa
4	Tipologia: stringa
5	Data: oggetto embedded
6	Path Documento: stringa
7	Tipologia Fonte: numero intero

Traduzione (schema in tabella 4.4 contiene documenti caratterizzati da un campo Messaggio (2), che riporta la parola o la frase nella lingua di default del sistema e un dizionario di traduzioni, costituito da coppie chiave-valore dove la chiave è la lingua e il valore la traduzione del Messaggio.

	Traduzione
1	ID: stringa alfanumerica
2	Messaggio: stringa
3	Linguaggio: dizionario

Tabella 4.4: Schema Traduzione

Tipo Elemento Descrittivo (schema in tabella 4.5) è la raccolta degli schemi che è possibile usare per creare Elementi Descrittivi. È in un certo senso una meta-collezione, perché riguarda la struttura di elementi appartenenti a un'altra collezione, quella degli Elementi Descrittivi. Questa complessità è dovuta al fatto che, al contrario di tutte le altre entità presentate fino a ora, lo schema degli Elementi Descrittivi può essere liberamente configurato dall'end-user e non consiste solamente in un insieme di campi per cui è possibile specificare o meno un valore. Il campo principale è *Elemento* (3), che contiene la struttura di un Elemento Descrittivo. Questo è una lista di oggetti embedded, in numero variabile. Ciascuno di essi ha quattro campi: Nome dell'elemento, Tipologia, Obbligatorietà e Ordine. L'insieme degli *Elementi* definisce la struttura di un Elemento Descrittivo sia a livello di documento MongoDb sia di form tramite cui verrà creato.

	Tipo Elemento Descrittivo
1	ID: stringa alfanumerica
2	Nome: stringa
3	Elemento: lista di oggetti embedded

Tabella 4.5: Schema Tipo Elemento Descrittivo

4.2 Infrastruttura Software

L'applicazione prevede due modalità di consultazione, cui corrispondono due diverse tipologie di utenti. Gli utenti registrati sono coloro che utilizzano il sistema come piattaforma di studio e ricerca e possono dunque accedere a tutte le funzionalità dell'Atlante. Questa parte della applicazione è definita *privata* ed è riservata a *professionisti del dominio* e *amministratori di sistema*. Ai *visitatori* è invece concessa la navigazione nella parte *pubblica*, che consiste principalmente di una interfaccia utente che presenta, sulla sinistra, una tabella riportante tutti i Palazzi inseriti nel sistema e, sulla destra, una mappa che ne mostra la distribuzione geografica. Per ciascun Palazzo, inoltre, è possibile consultare una scheda descrittiva. Alcuni esempi delle funzionalità delle piattaforma verranno mostrati nel prossimo paragrafo. La divisione in due parti è stata fortemente incentivata dall'uso di Django, che prevede la creazione automatica di una sezione amministrativa in cui è possibile, tra le altre cose, gestire in modo semplice utenti e i loro privilegi.

La figura 4.4 mostra la mappa di navigazione; ogni rettangolo costituisce una pagina dell'Atlante Digitale. Ciascuna di esse è associata a un Template, un documento in formato HTML che funge da "modello" e che viene opportunamente compilato a seconda delle necessità. I Template contengono blocchi che sono stati definiti in altri documenti, come "base.html" e "private_base.html" poiché alcuni elementi, come la barra di navigazione, sono richiesti in ogni scheda; è dunque comodo scriverli un'unica volta in un singolo file ed evitare di replicare il codice. Oltre al Template ogni pagina possiede uno script scritto in JavaScript. Al suo interno vengono usate le librerie di supporto, quali JQuery, Leaflet e TinyMCE, per la gestione di eventi, mappe e testo formattato. Lo script fornisce dinamicità alla pagina, definendo funzioni che rispondono all'interazione dell'utente con gli elementi del documento HTML. Le operazioni che l'utente può effettuare possono essere divise in due gruppi: navigazione tra schede diverse e richieste HTTP asincrone, dette richieste AJAX. Nel primo caso l'URL della risorsa identifica una funzione della View che restituisce il Template associato alla pagina richiesta. Ricevuto il Template, il web browser dell'end-user lo renderizza, aggiornando la pagina. Questa operazione è detta sincrona perché a fronte della richiesta l'utente aspetta la risposta senza poter continuare a usare il sito. Nelle richieste asincrone, invece, l'URL conduce a una funzione della View che estende la classe APIView, implementazione REST in Django. L'APIView non restituisce un Template ma un documento JSON, che lo script riceve e processa in background, senza interferire con il comportamento della pagina, che viene aggiornata senza essere ricaricata.

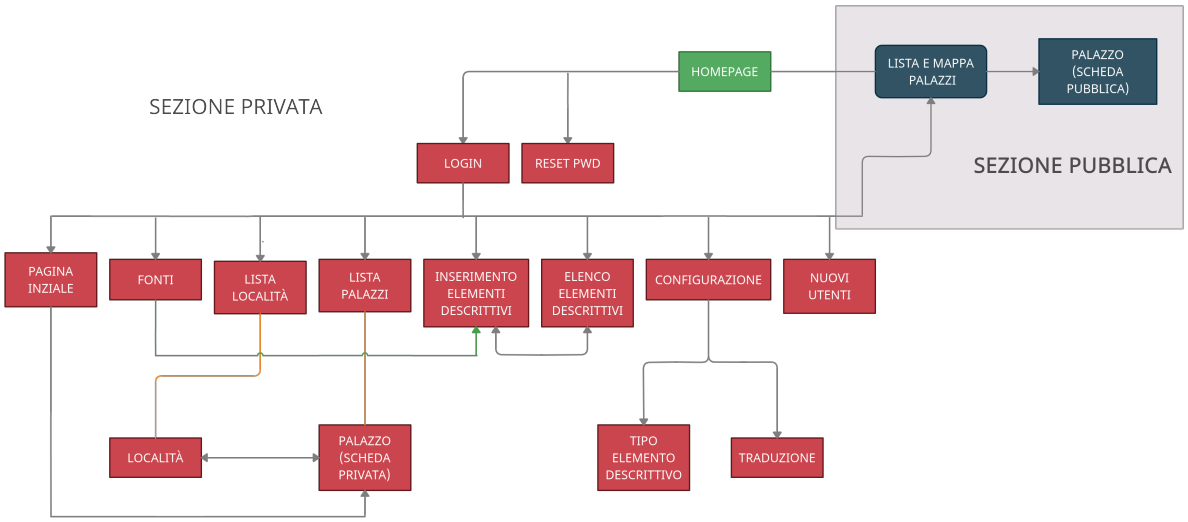

Figura 4.4: Mappa del sito web

4.3 Funzionalità della piattaforma

Dalla figura 4.4 è chiaro che la maggior parte delle funzionalità dell'applicazione riguarda la parte privata del sistema, accessibile a un numero esiguo di utenti. Queste condizione, però, non ha determinato una scarsa attenzione nei confronti della User Experience (spesso abbreviata in UX). Tipicamente gli strumenti riservati a specialisti sono graficamente più essenziali rispetto a quelli pensati per un pubblico di massa, che tra migliaia di applicazioni funzionalmente simili sceglierà quella più "bella" e innovativa a livello di design. Per questa ragione la parte privata del sito è stata costruita per favorire gli aspetti più pratici della UX. Il flusso di navigazione è pensato per spostarsi in modo logico tra le schede anche senza ricorrere alla barra di navigazione. Le entità fortemente interdipendenti, come ad esempio Località ed Edifici Comunali, sono visualizzate in un'unica scheda per dare una più ampia visione di insieme. L'utilizzo di Bootstrap ha inciso molto sull'usabilità del sistema: molte pagine presentano componenti della libreria che uniscono stile e funzionalità. Un esempio è *l'accordion*, menù costituito da più schede (che possono essere aperte e chiuse "come una fisarmonica") che si è rivelato molto utile nei contesti in cui ci sono tante informazioni da mostrare ma non si vuole appesantire l'interfaccia. Grazie all'accordion l'utente può selettivamente scegliere quali dati visualizzare. La scheda principale della parte pubblica, mostrata in figura 4.5, è costituita da una **tabella** e una **mappa**.

Esse presentano la stessa informazione, i Palazzi, in due modi diversi: ogni Palazzo è una riga nella tabella e un marker sulla mappa. Il colore dei marker varia a seconda del valore del campo Riconoscibilità (vedere tabella 4.1). Se un Palazzo è definito come poligono i suoi contorni saranno mostrati su mappa e, nel loro centro di questi verrà posizionato il marker. Cliccando con il mouse sul marker viene aperto un pop-up che mostra il nome del Palazzo e la Località. Il nome è un ipertesto che porta alla scheda pubblica del palazzo. Al variare del livello di zoom sulla mappa i Palazzi sono raggruppati in cluster il cui colore ne mostra la densità in quella specifica zona.

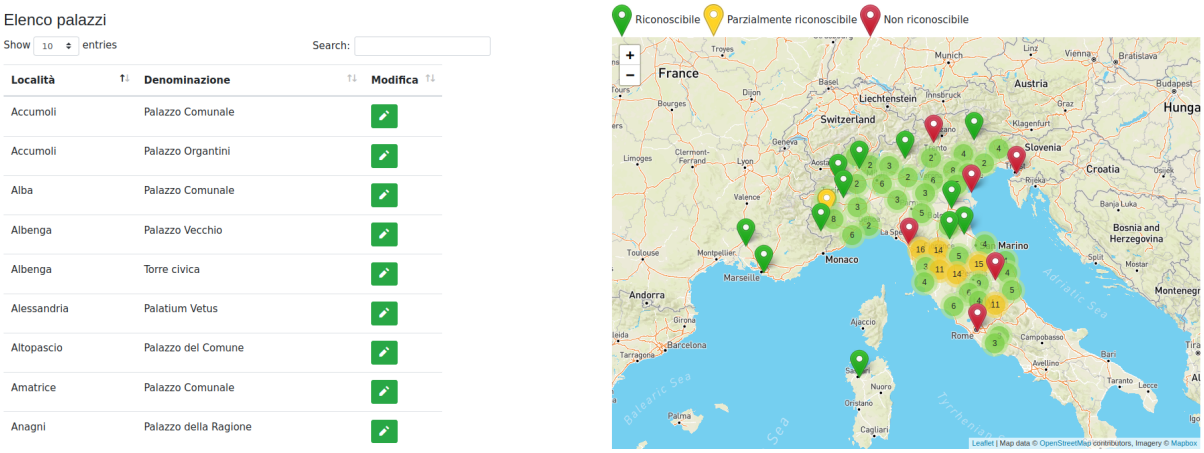

Figura 4.5: Mappa del sito web

Mappa e tabella sono legate in modo tale da presentare la stessa informazione a fronte di un'azione svolta dall'utente. In particolare se quest'ultimo effettua una ricerca sulla tabella la mappa mostrerà solamente i marker relativi ai risultati della ricerca e sarà centrata in un punto medio (stabilito prendendo il minimo e il massimo di longitudine e latitudine di tutti i Palazzi). Questa situazione è mostrata in figura 4.6.

Similmente, se l'utente si sposta sulla mappa la tabella sarà dinamicamente aggiornata per mostrare solamente i Palazzi inclusi nella porzione visibile di mappa. Durante lo sviluppo di questa funzionalità il comportamento della mappa è stato più volte cambiato. In una prima versione i marker dei Palazzi che non comparivano nei risultati di una ricerca su tabella venivano colorati di grigio e resi semi-trasparenti. Ora, invece, gli stessi marker vengono cancellati dalla mappa. Entrambi gli approcci hanno i loro vantaggi: continuare a mostrare i marker sebbene essi non siano inclusi nei risultati di ricerca può fornire informazioni riguardo alla vicinanza a quelli mostrati; rimuoverli alleggerisce la visualizzazione. Cliccando su una riga della tabella, inoltre, viene eseguito uno zoom sulla mappa per mostrare il Palazzo di interesse, il cui pop-up viene aperto.

Altra funzionalità importante è l'integrazione con **Zotero**, software per la gestione di riferimenti bibliografici e dei materiali ad essi correlati. Zotero è uno strumento open-source usato per collezionare, organizzare, citare e condividere materiale di ricerca. Questo può essere raccolto da un sito web, organizzato tramite tag e collezioni configurabili, arricchito con una bibliografia, sincronizzato tra i vari device e facilmente condiviso tra i vari utenti [48]. Poiché il sistema sviluppato si propone di essere una piattaforma versatile in grado di immagazzinare, gestire e modificare informazioni eterogenee sugli edifici storici, era logico pensare di renderlo compatibile con strumenti molto utilizzati nel dominio della ricerca e dell'architettura. In particolare con Zotero è possibile creare una collezione di fonti bibliografiche ed esportarle sotto forma di file JSON, la cui struttura è stata già descritta nell'ambito dei documenti MongoDB. A questo punto è possibile accedere alla scheda Fonti dell'Atlante Digitale e importare il file stesso. Questo verrà analizzato, fonte per fonte, per vedere quali di esse presentano i campi obbligatori richiesti e sono dunque inseribili nel sistema. Terminata l'analisi, la collezione Fonti sarà arricchita da tanti documenti quanti sono le fonti del file JSON aventi i campi richiesti.

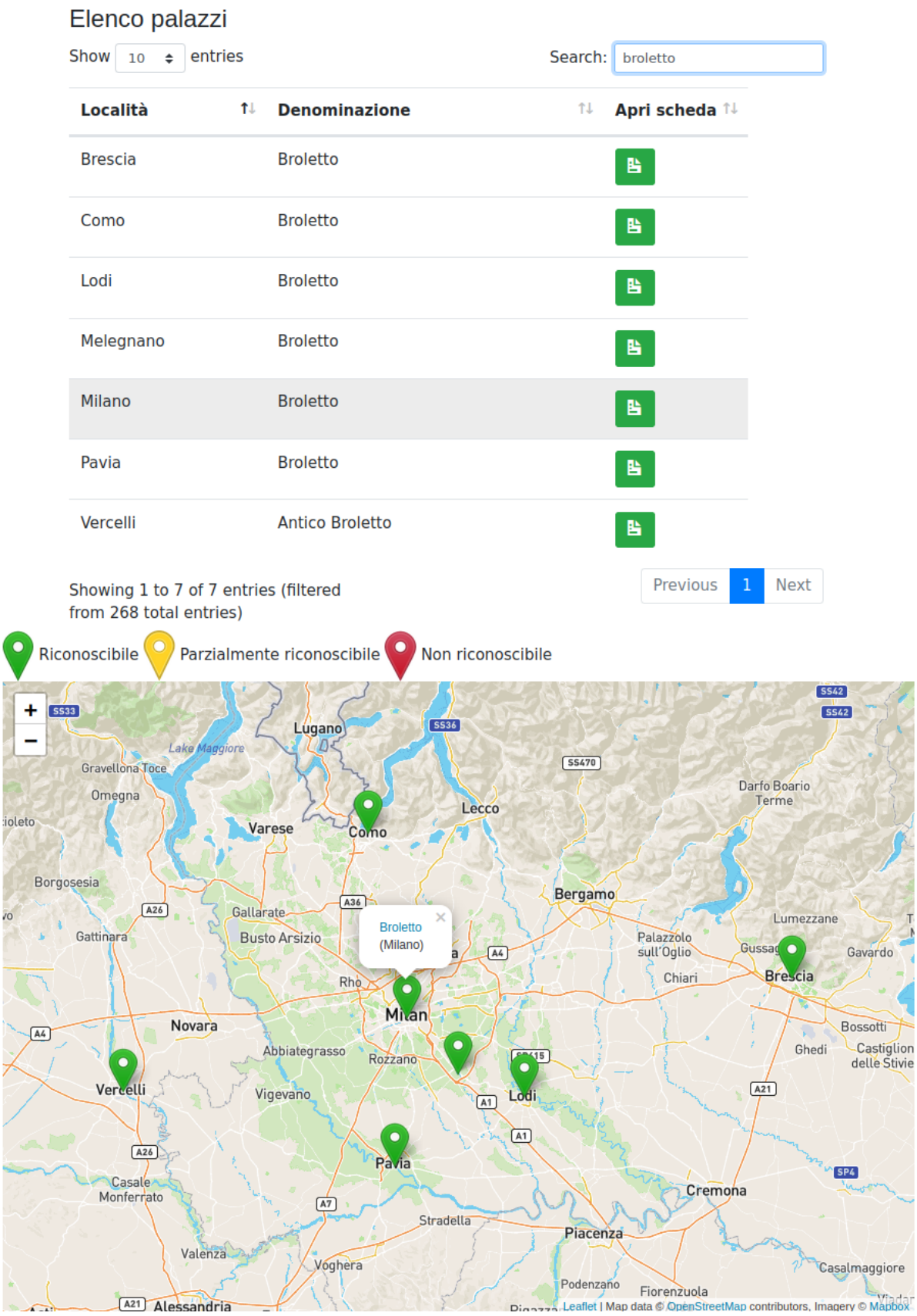

Figura 4.6: Risultato della ricerca su tabella

Alle altre funzionalità si è più o meno direttamente già accennato nei capitoli e paragrafi precedenti. La possibilità di creare e modificare gli Elementi Descrittivi dalla scheda Configurazione è sicuramente uno dei punti di forza del sistema, in quanto consente agli amministratori del sistema di svincolarsi dal lavoro degli sviluppatori. Dalla stessa scheda Configurazione è anche possibile compilare il dizionario di traduzione in varie lingue, indispensabile per l'internazionalizzazione del sistema. Per ogni Palazzo è possibile creare una scheda descrittiva utilizzando il plugin TinyMCE. La scheda può contenere immagini e testo formattato. In ultimo vanno ricordate tutte quelle operazioni che hanno a che vedere con la gestione degli utenti: login, logut, cambio password, registrazione e divisione in gruppi.

Capitolo 5

Casi d'uso

In questo capitolo verranno illustrati e descritti alcuni dei Casi d'Uso più significativi del sistema, riguardanti non solo la creazione e gestione delle entità come Località, Fonti e Palazzi ma anche gli aspetti di configurazione ed amministrazione. Le sottosezioni sono organizzate in modo tale da creare un percorso che va dalla creazione dell'entità di base alla visualizzazione d'insieme di tutti gli elementi ad essa correlati. In particolare i paragrafi da 1 a 3 riguardano Località, Edifici Comunali e Palazzi; da 4 a 6 Fonti ed Elementi Descrittivi.

Le funzionalità offerte dal sistema seguono lo stesso pattern di interazione tra frontend e backend. Ogni pagina è costituita da elementi HTML su cui è possibile compiere un'azione, chiamata evento. Il click su un tasto "Salva" in una pagina contenente un form è l'esempio più semplice, sebbene non tutti gli eventi richiedano l'azione diretta dell'utente. La reazione a questi eventi è una funzione definita nello script JavaScript associato alla pagina HTML. Se la gestione dell'evento richiede la comunicazione con il server la funzione conterrà una richiesta HTTP asincrona AJAX. La richiesta AJAX è rivolta a un URL e può contenere dati nel formato JSON. Quest'ultimi vengono utilizzati o per interrogare la base di dati (che però non subisce modifiche) o analizzati per essere opportunamente salvati nel database (si verifica un cambio di stato). Nel primo caso viene usato il metodo GET, nel secondo POST. Come già ampiamente discusso, a questo punto l'URL Dispatcher di Django si occupa di invocare la View deputata alla gestione della richiesta, che risponderà con un Template o, più spesso, con un file JSON. In figura 5.1 è riportato il diagramma dei casi d'uso del sistema. Da questo è possibile notare la suddivisione dei compiti tra le varie tipologie di utenti.

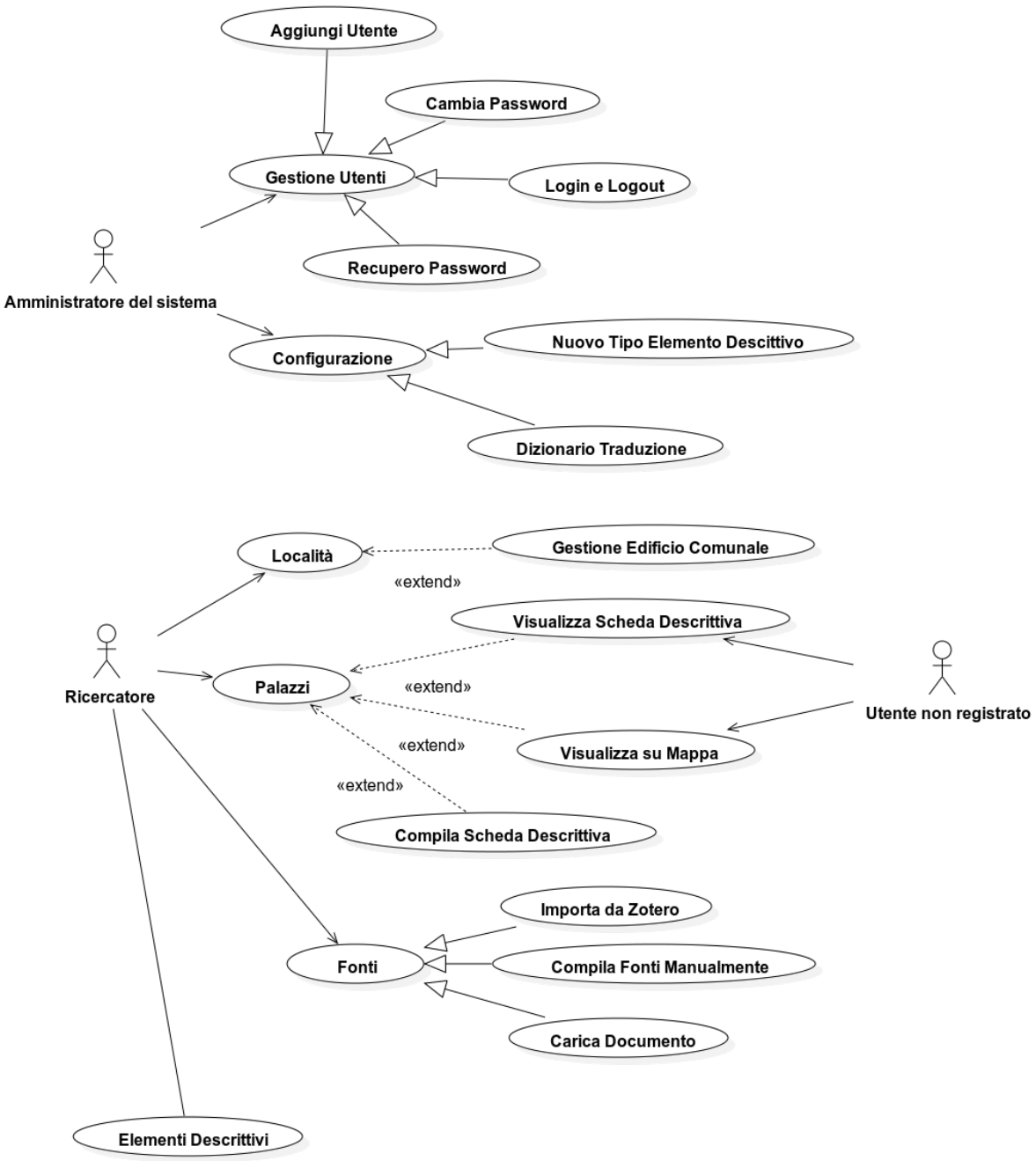

Figura 5.1: Diagramma dei Casi d'Uso del sistema

5.1 Creazione e gestione di una Località

Il primo gruppo di Casi d'Uso è relativo all'entità centrale dell'Atlante, i Palazzi. Prima di definire un Palazzo è tuttavia necessario creare una **Località** che lo contenga.

Dopo aver effettuato il login ed essere entrati nella parte privata del sito si accede alla scheda *Località* della barra di navigazione. Questa mostra una tabella che riporta tutte le località già presenti nel sistema. A questo punto si può scegliere se modificarne una già esistente cliccando sull'icona a forma di matita oppure crearne una nuova, premendo ***Aggiungi Località***. Il click su questo pulsante porta all'apertura di una finestra modale che chiede all'utente di inserire il nome della nuova Località (figura 5.2.

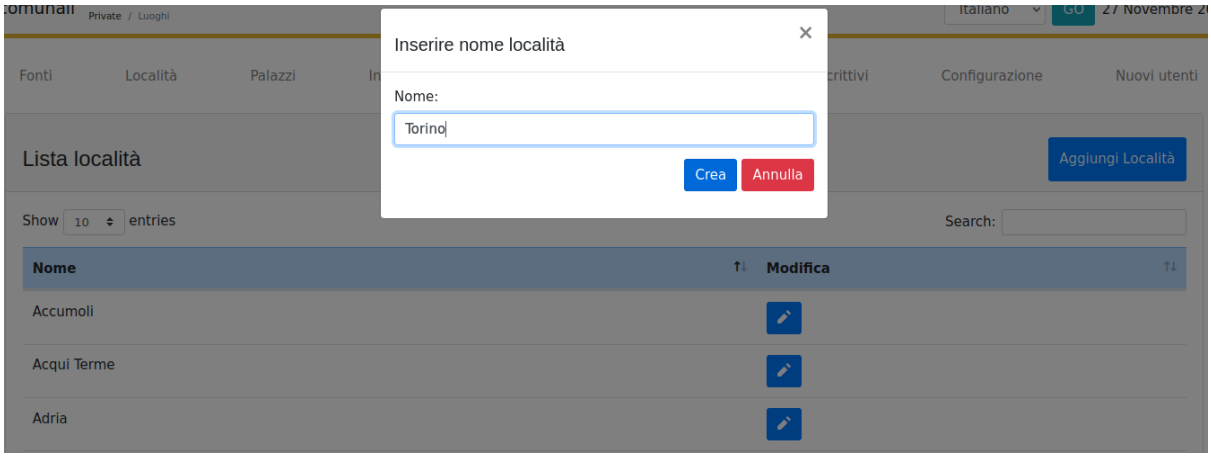

Figura 5.2: Creazione di una nuova Località

Deciso il nome, la pressione del tasto ***Crea*** conduce alla pagina dedicata alla singola Località, appena creata e inserita nel database. La scheda presenta informazioni relative non solo alla Località in sé ma anche ai Palazzi e gli Edifici Comunali ad essa associati. Per questioni di visualizzazione nella figura 5.3 è tuttavia mostrata solo la parte relativa alla Località. Una visione globale sarà offerta dai prossimi casi d'uso

Le informazioni che caratterizzano la Località sono il nome, le denominazioni precedenti e la prima attestazione dell'edificio comunale. Il campo *Denominazioni Precedenti* è realizzato mediante il plugin **amsify-suggestags** che permette di convertire un semplice input testuale in una lista di etichette, o tag. *Prima attestazione dell'organismo comunale* fa riferimento alla data di istituzione del comune (e non a quella di fondazione) e può avere tre formati: data completa, mese e anno, solo anno. Oltre a queste proprietà è possibile disegnare i contorni della Località tracciandoli sulla mappa mostrata in figura 5.3. Per abilitare questa funzione è dapprima necessario premere il pulsante ***Modifica Perimetro***. Come già accennato in precedenza, la Località può essere indicata da un punto o da un poligono (anche irregolare).

Compilata la scheda è possibile salvare il form tramite il pulsante ***Salva*** o resettare i campi testuali mediante ***Resetta il Modulo***. Il salvataggio dei dati è consentito solo se è stato compilato il campo Nome e la Località è stata segnata su mappa.

La Località può essere eliminata tramite il tasto ***Elimina Località***. Questa operazione è tuttavia possibile solo se non esistono Palazzi o Edifici Comunali ad essa associati.

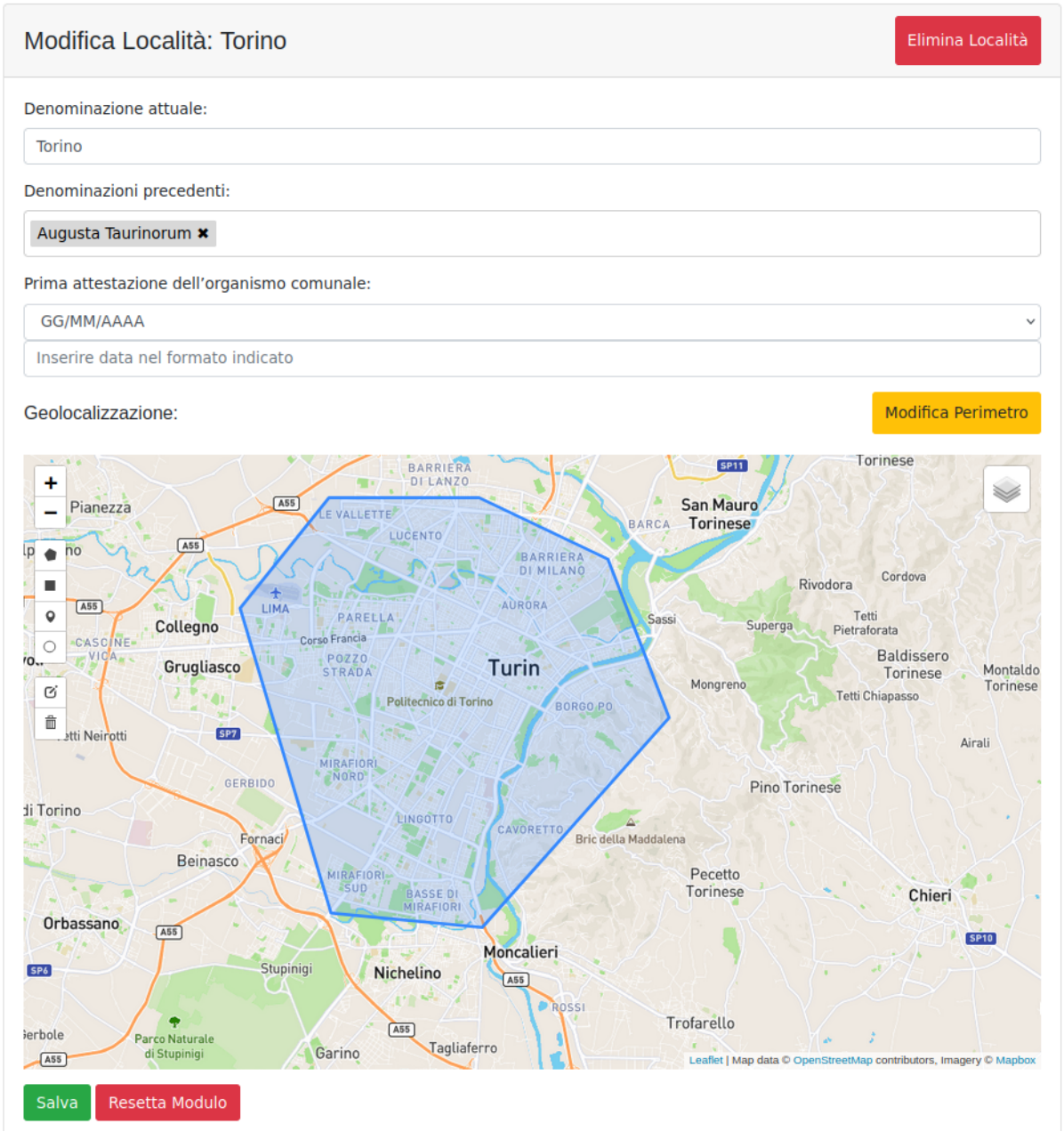

Figura 5.3: Pagina Località (solo parte sinistra)

5.2 Creazione e gestione di un Edificio Comunale

Nella stessa pagina della Località è presente la sezione dedicata agli Edifici Comunali, realizzata come menù ad *accordion*. Il menù presenta due schede: ***Mostra Edifici Comunali*** e ***Aggiungi Edificio Comunale*** (mostrata in figura 5.4). La prima mostra in forma tabulare gli Edifici già aggiunti, la seconda conduce al form di creazione. Quest'ultimo prevede quattro campi: Denominazione attuale o consolidata, Funzione, Identificazione e Denominazione, Altro. ***Denominazione Attuale o consolidata*** è il nome dell'edificio; la ***Funzione*** è selezionata mediante un menù a tendina (componente **select-picker** di boostrap); il campo ***Identificazione e Denominazione*** permette di definire fino a cinque Identificazioni dell'edificio, ciascuna caratterizzata da 1) Un menù a tendina che riporta la natura dell'edificio (Casa di privati, Spazio vescovile, Piazza o mercato etc.) 2) Una breve descrizione e 3) Una Data; il campo testuale ***Altro*** serve a raccogliere tutte le informazioni che non sono legate ai campi precedenti.

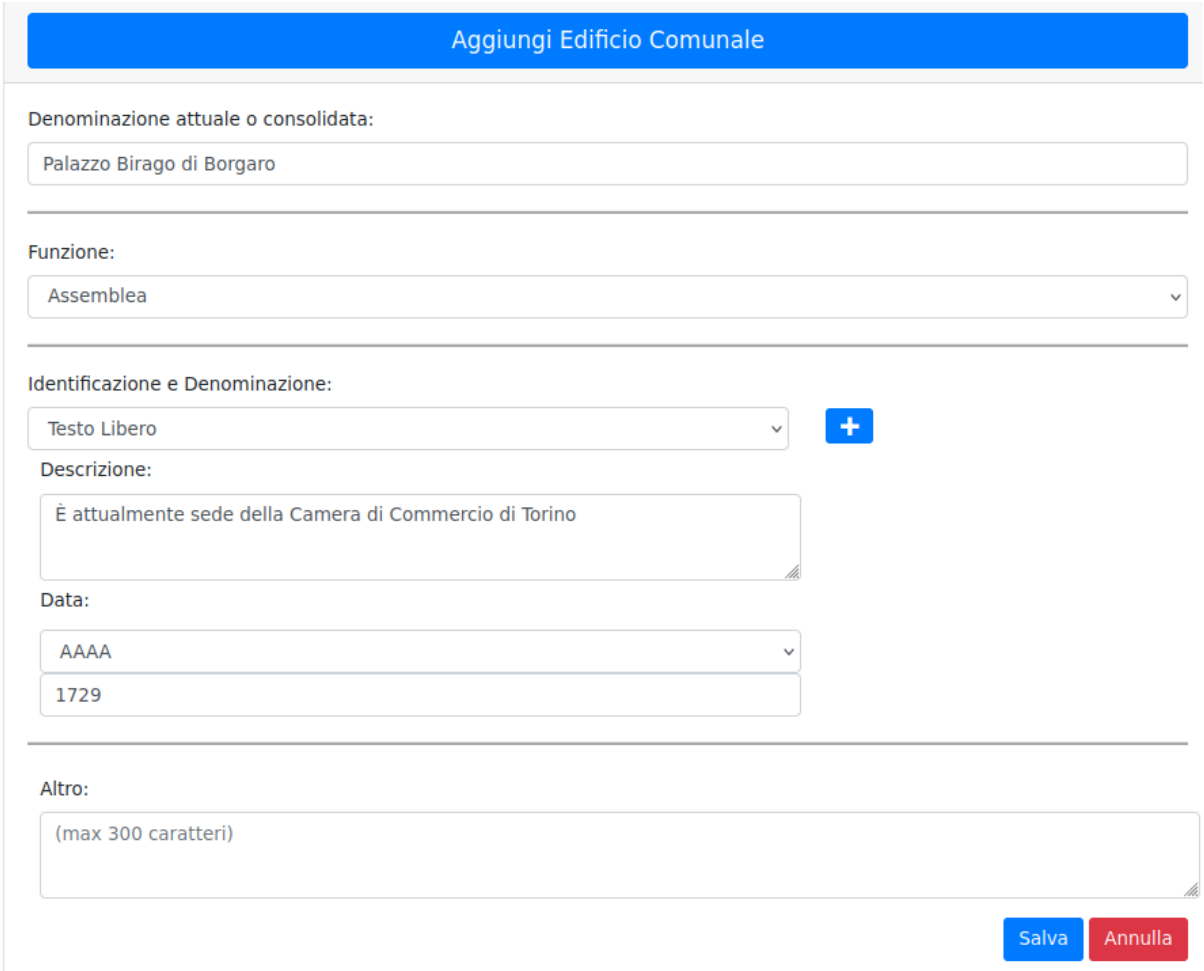

Figura 5.4: Creazione di un edificio comunale

Sulla stessa mappa riportante la Località è possibile definire la posizione dell'Edificio Comunale. Quando viene aperto il menù di creazione dell'Edificio, i tasti ***Modifica Perimetro***, ***Salva*** e ***Resetta Modulo***, tutti funzionalmente legati alla Località, vengono disabilitati per evitare di creare confusione. Il plugin della mappa consente di lavorare su più livelli, o layer: Località ed Edifici Comunali hanno due layer diversi, e il passaggio da uno all'altro avviene all'apertura

e alla chiusura della scheda ***Aggiungi Edificio Comunale***. A differenza della Località la posizione e il perimetro dell'Edificio Comunale non sono obbligatori; un pop-up si occupa di avvertire l'utente della potenziale creazione di un Edificio senza contorni. Questa scelta è stata presa per due motivi: in primis Edificio Comunale non è un'entità di base come Località e in secondo luogo per permette di creare Edifici la cui posizione è sconosciuta o ignota.

Una volta compilato il form di creazione dell'Edificio, il tasto ***Salva*** si occupa di formulare la richiesta di salvataggio al backend. Terminato il salvataggio, una riga viene aggiunta alla tabella degli Edifici Comunali. Da questa è possibile sia modificare l'Edificio già esistente sia eliminarlo. La figura 5.5 mostra il marker, con relativo pop-up, di un edificio comunale.

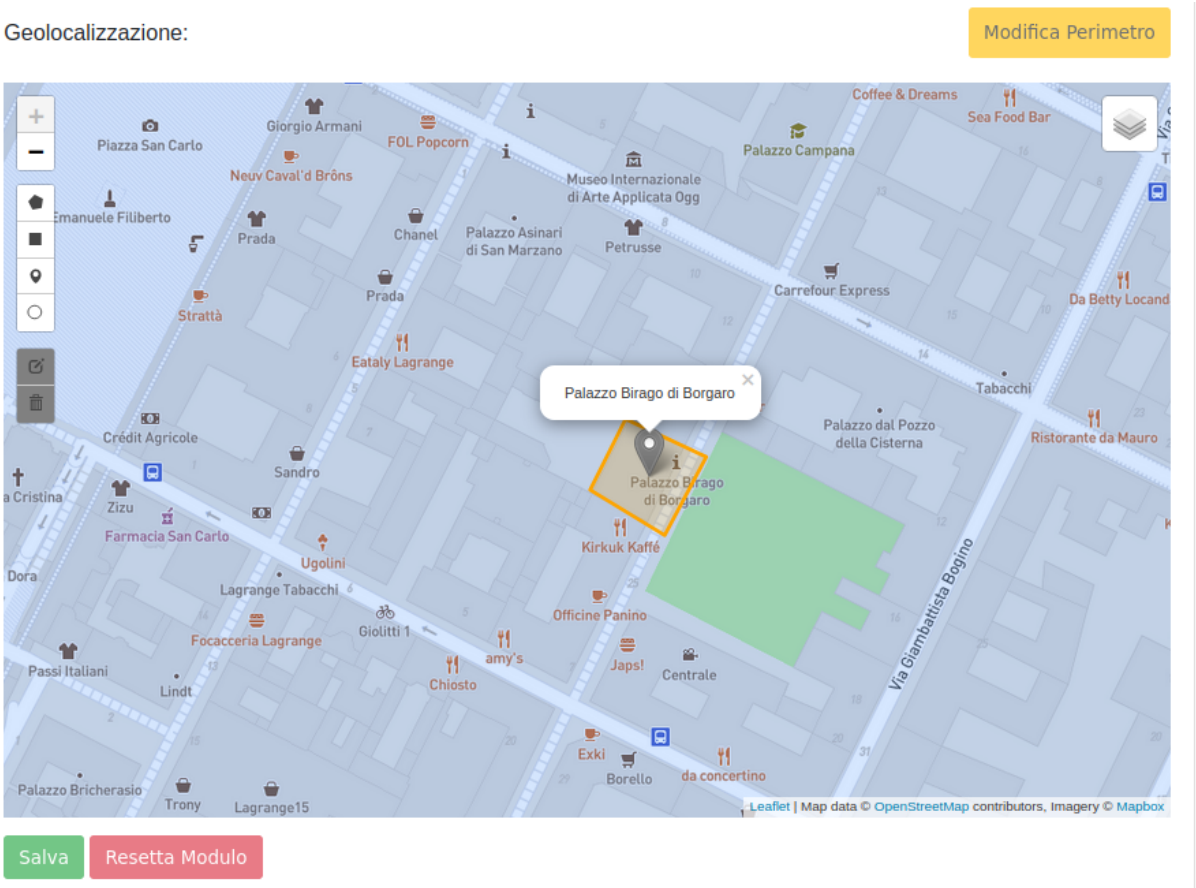

Figura 5.5: Marker edificio su Mappa

5.3 Creazione e gestione Palazzo, con annessa sintesi

A differenza degli Edifici Comunali i Palazzi hanno una scheda dedicata. Ci si accede dalla barra di navigazione e presenta una tabella identica a quella della scheda Località, ovvero riportante tutti i Palazzi già inseriti nel sistema e contenente un bottone ***Aggiungi Palazzo***, come mostrato in figura 5.6.

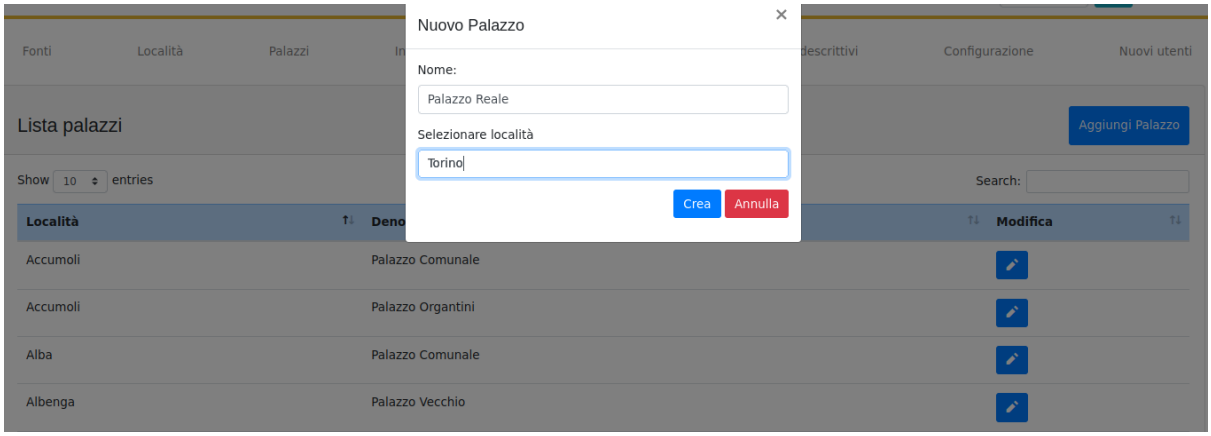

Figura 5.6: Lista Palazzi

Scelto il nome e premuto il tasto ***Crea*** si viene reindirizzati alla pagina privata del singolo Palazzo. Questa presenta un menù ad accordion in cui la scheda superiore mostra il form di modifica del Palazzo e quella inferiore la Sintesi. La figura 5.7 offre una visione d'insieme.

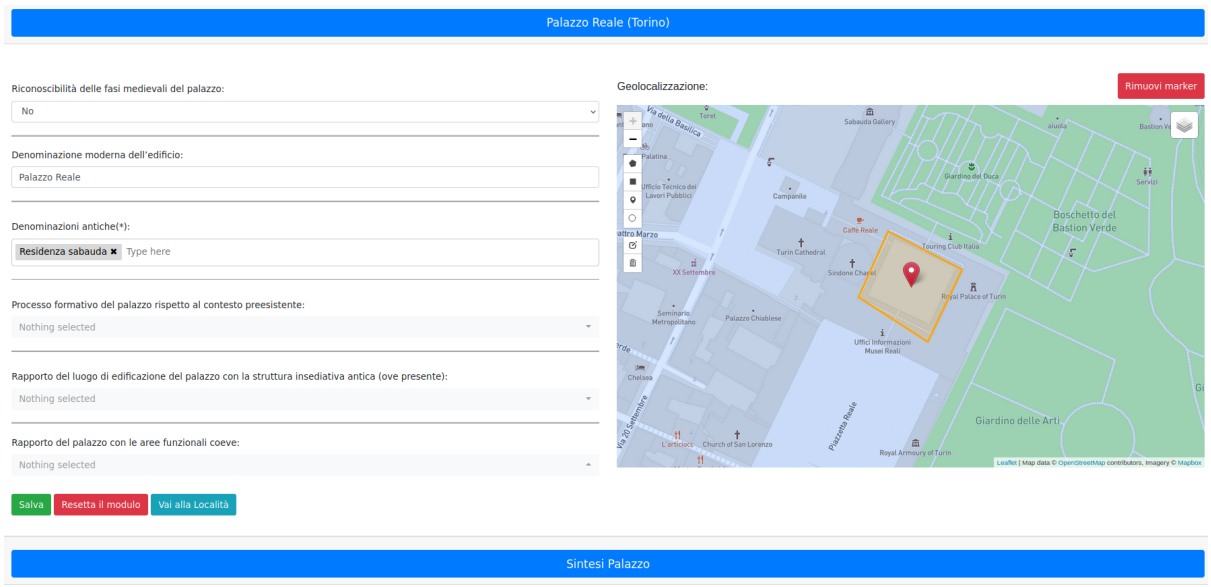

Figura 5.7: Pagina privata del singolo Palazzo

Sebbene la risoluzione non permetta di leggere agevolmente i vari campi, la modalità di gestione di un Palazzo è del tutto simile a quella di Località ed Edifici Comunali: sono presenti un form e una mappa su cui segnare l'edificio. Poiché il Palazzo è l'entità concettualmente più importante dell'Atlante, il form è ricco di campi che lo caratterizzano. Non vale la pena descriverli tutti, anche perché a livello HTML non sono presenti campi di input diversi da quelli già presentati. Una piccola differenza rispetto agli Edifici Comunali è il colore del marker visualizzato su mappa, che varia a seconda della ***Riconoscibilità***

La sezione ***Sintesi Palazzo*** permette di compilare una scheda descrittiva mediante un ricco editor di testo, implementato dal plugin **TinyMCE**. Un esempio di scheda è mostrato in figura 5.8. È possibile associare a ogni Palazzo due Sintesi, una in Italiano e una in Francese.

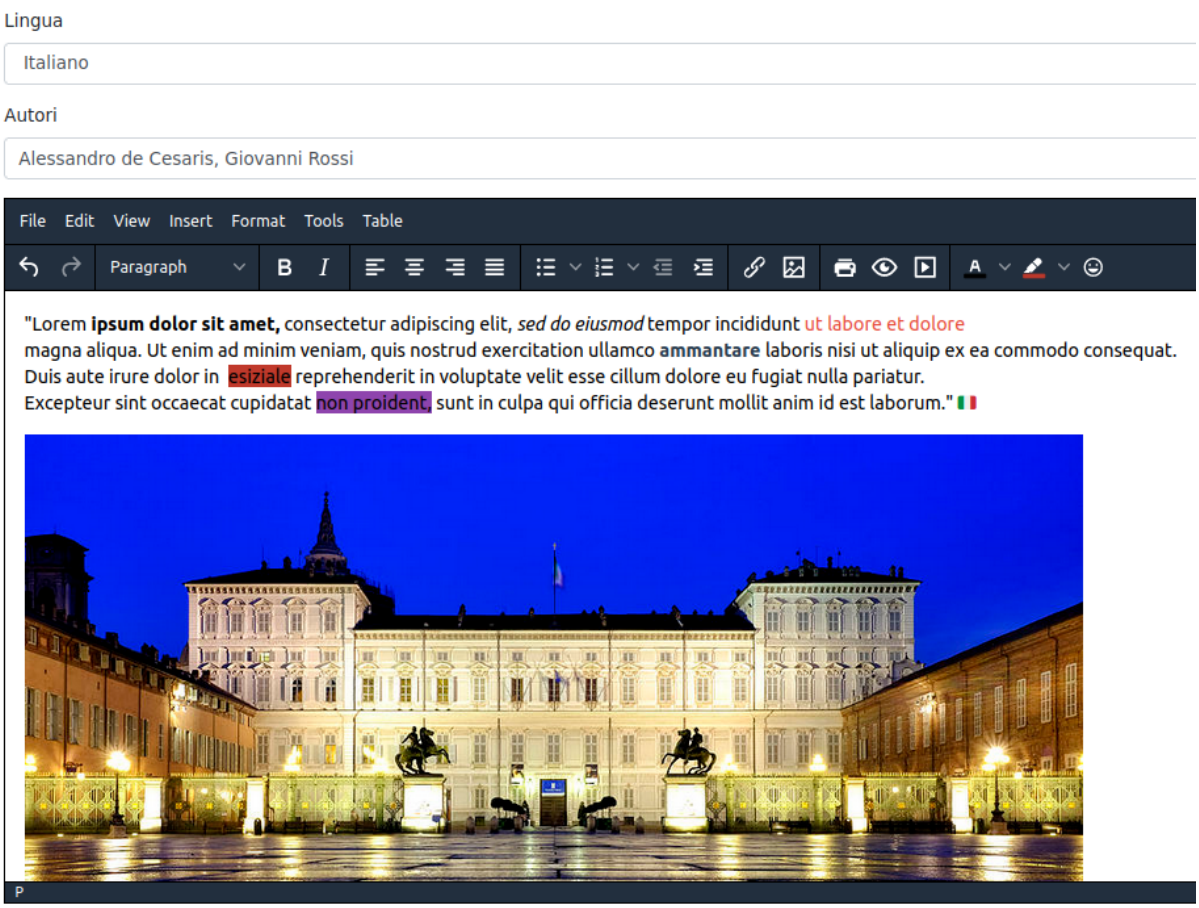

Figura 5.8: Esempio di Sintesi Palazzo

Rimanendo sulla stessa pagina è possibile accedere alla Località associata al Palazzo tramite il pulsante ***Vai alla Località***. L'immagine 5.9 mostra come si presenta la scheda Località dopo aver creato un Edificio Comunale e un Palazzo: l'informazione relativa alle tre entità trattate fino a ora è tutta contenuta in una singola scheda. Tramite i due menù ad accordion è possibile visualizzare in forma tabulare quali Palazzi ed Edifici sono presenti sulla mappa. Da notare che i marker utilizzati e i loro colori sono diversi a seconda della tipologia di edificio.

Dalla scheda Località non solo è possibile accedere alla scheda del Palazzo già creato per modificarlo ma anche aggiungerne direttamente uno nuovo, cliccando il tasto ***Aggiungi Palazzo***.

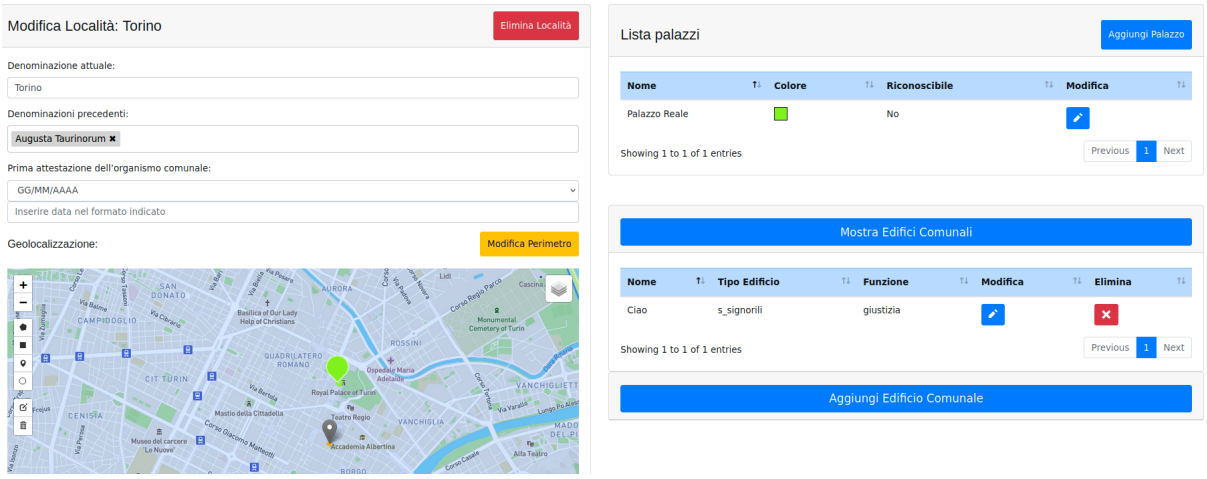

Figura 5.9: Riepilogo scheda Località

5.4 Creazione nuovo Tipo Elemento Descrittivo

Questo paragrafo inaugura quella parte dei casi d'uso relativa alla gestione di Elementi Descrittivi e Fonti. Per evitare confusione, va ricordato che Tipo Elemento Descrittivo ed Elemento Descrittivo sono due entità diverse: la prima definisce la struttura della seconda. par Come primo passo è dunque necessario creare un nuovo **Tipo Elemento Descrittivo**, operazione che è possibile svolgere accedendo alla scheda ***Configurazione*** (1) della barra di navigazione e selezionando ***Tipo Elementi Descrittivi*** (2). Le tipologie già create sono consultabili nella tabella (4) della figura 5.10.

Figura 5.10: Scheda Configurazione relativa agli Elementi Descrittivi

Selezionando ***Nuovo tipo di elemento*** (3) appare la sezione che permette di creare la struttura dell'Elemento Descrittivo. La struttura è formata da varie componenti; ciascuna di essa descrive un singolo campo di input del form. Per ogni componente è necessario specificare Nome (2), Tipologia (3), Obbligatorietà (4) e Ordine (5). La Tipologia determina la proprietà *type* del campo: text, textarea, number, boolean o date. L'Obbligatorietà segna la presenza o meno della proprietà *required*, l'Ordine la posizione che il campo assumerà nel form finale. È possibile aggiungere componenti premendo il tasto ***Aggiungi Elemento Form*** (1)

Effettuato il salvataggio, nel database sarà presente un documento nel formato mostrato nella tabella 4.5 in cui il campo Elemento conterrà tutte le componenti definite poc'anzi. Questo documento sarà successivamente utilizzato per creare il form degli Elementi Descrittivi di tipo "Giardino", stando all'esempio mostrato in figura 5.11.

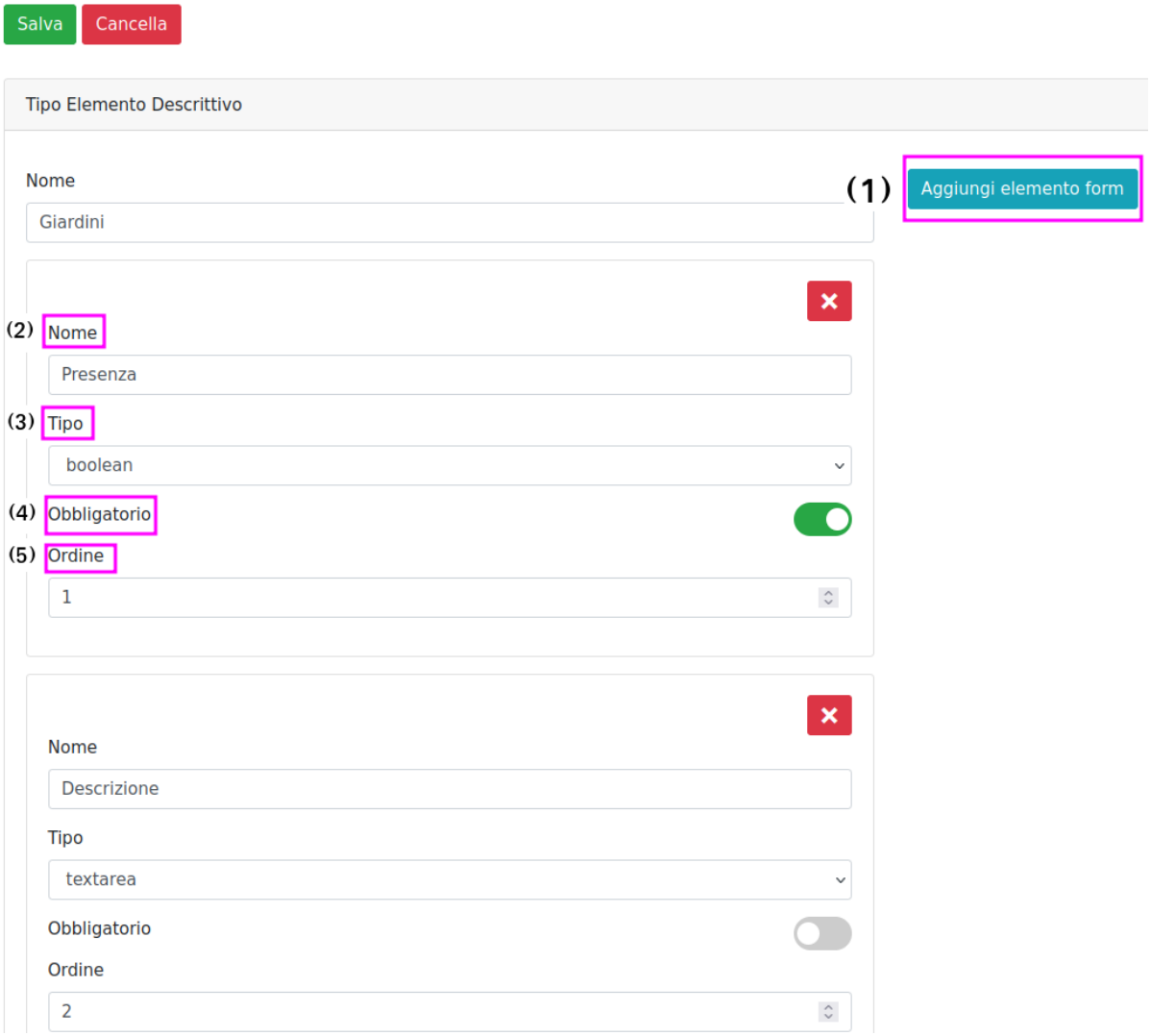

Figura 5.11: Creazione di una nuova tipologia di Elemento Descrittivo

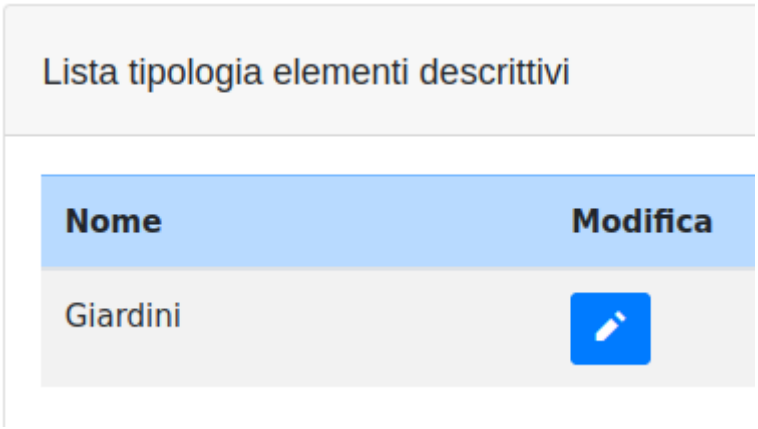

Figura 5.12: Tabella degli Elementi Descrittivi

5.5 Creazione manuale di una Fonte, con annesso caricamento documento allegato

Come è possibile osservare dal data model in figura 4.3 un Elemento Descrittivo è sempre legato a una Fonte. A differenza dei Tipi di Elemento, oggetti del caso d'uso precedente, le Fonti non sono configurabili dall'end-user ma definite staticamente. Non esiste dunque una collezione "Tipo Fonte" nel database. Questa differenza è dovuta al fatto che la variabilità di una Fonte e delle sue caratteristiche è minore rispetto a quella di un Elemento Descrittivo

Una Fonte può essere inserita manualmente, ovvero compilata in un form appositamente dedicato o importata da Zotero sotto forma di file JSON. Questo Caso d'Uso mostra la prima modalità, quella della creazione manuale.

Anche la scheda Fonti è accessibile dalla barra di navigazione. Alla sua apertura un menù a tendina propone le varie tipologie di Fonti. Tra queste esistono differenze non solo in termini di campi ma anche di funzionalità. Fonti Bibliografiche è la tipologia in cui vengono mappate le fonti Zotero, dunque il menù di importazione è accessibile solo selezionando questa Fonte specifica. Alcune Fonti sono univocamente identificabili da Autori e Titolo, altre dalla Segnatura Archivista. Determinate tipologie prevedono inoltre la possibilità di allegare un file con estensioni .pdf, .jpeg o .tiff. Tra di esse c'è *Letteratura Grigia*, scelta per questo caso d'uso.

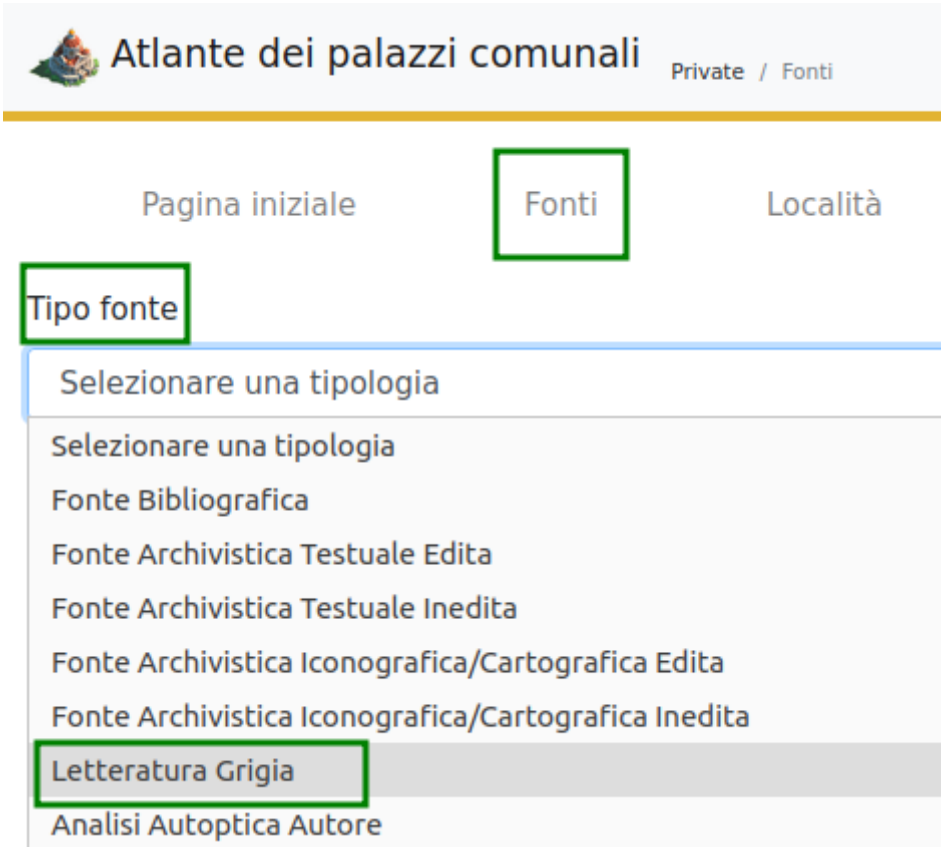

Figura 5.13: Creazione di una nuova Fonte della tipologia Letteratura Grigia

Selezionata la voce ”Letteratura Grigia” dal menù a tendina (figura 5.13) vengono mostrati i campi del form. Tra di essi merita considerazione **Autori** poiché la sua struttura ha implicazioni sul modo in cui il sistema tratta i dati e si occupa di individuare duplicati. Per ogni Fonte che presenta questo campo è possibile esprimere nome e cognome di uno o più autori, per un massimo di cinque. Quando una nuova Fonte viene inserita nel sistema una funzione del backend controlla che non ne esistano già altre aventi lo stesso titolo e almeno uno degli autori in comune.

Il modo in cui è gestito il caricamento di un file è simile a quello che lega un Palazzo alla sua Sintesi: prima viene creata l'entità di base e solo successivamente integrata l'informazione aggiuntiva. Come conseguenza al salvataggio della Fonte compare il campo contrassegnato con il rettangolo rosso nella figura 5.15 che permette di allegare un documento.

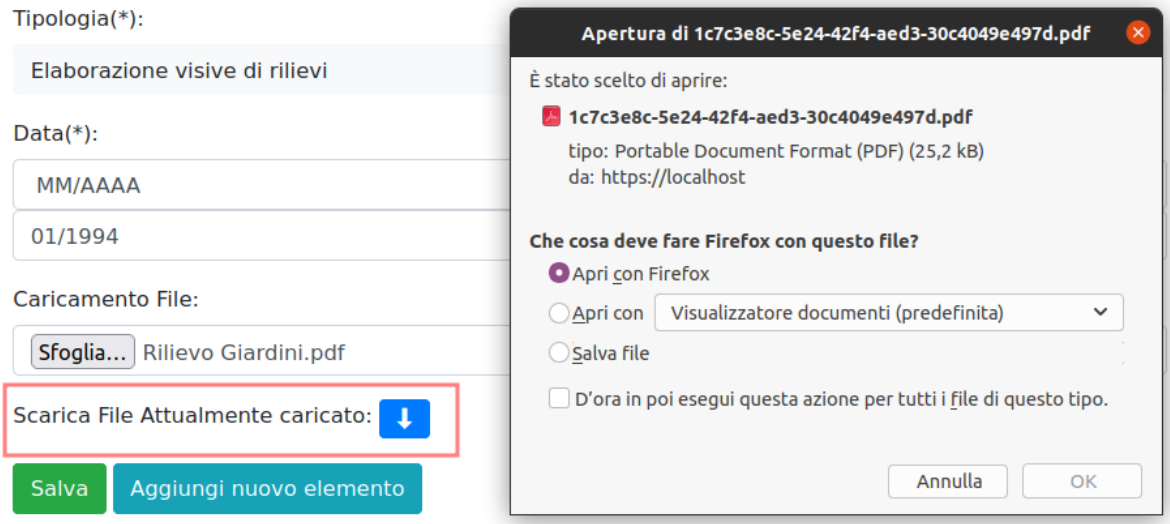

Figura 5.14: Download documento allegato a una Fonte

Cliccando su ***Sfoglia*** viene aperta la ben nota finestra che permette di selezionare un file con l'estensione corretta nel file system dell'end-user. Al caricamento del file nel browser un pop-up di conferma avvisa l'utente che eventuali file già presenti verranno sovrascritti. Se l'utente conferma la scelta una richiesta AJAX contenente il documento caricato e l'identificativo della Fonte viene inviati al backend. La funzione appositamente definita nella View si occupa di gestire il salvataggio dell'allegato nel file system del server, in un percorso stabilito in un file di configurazione del framework, *settings.py*. Terminata l'operazione il backend comunica l'esito rispondendo alla richiesta di salvataggio con un messaggio nel formato JSON. Se tutto è andato a buon fine il form della Fonte mostrerà un ulteriore campo, quello relativo al download del documento appena caricato (figura 5.14)

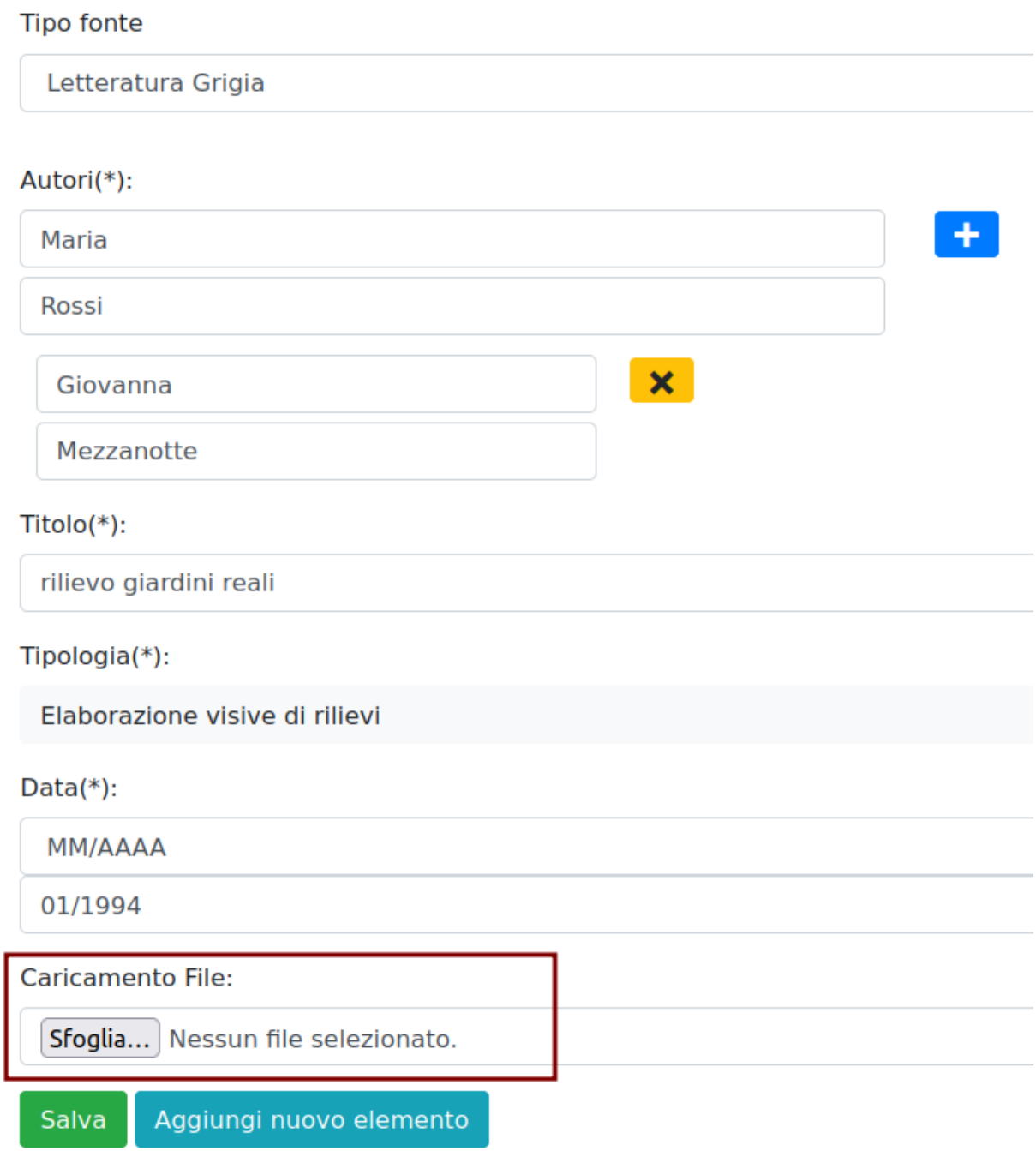

Figura 5.15: Form di creazione Fonte

5.6 Importazione di un documento da Zotero

Se dal menù a tendina mostrato nella figura 5.13 si sceglie l'opzione **Fonte Bibliografica** la finestra che compare presenta una sezione dedicata all'importazione di un file nel formato JSON. Questo file non è altro che il risultato dell'esportazione di un gruppo di fonti bibliografiche presenti nel software Zotero, di cui si è parlato nel Capitolo 4. L'utente ha dunque

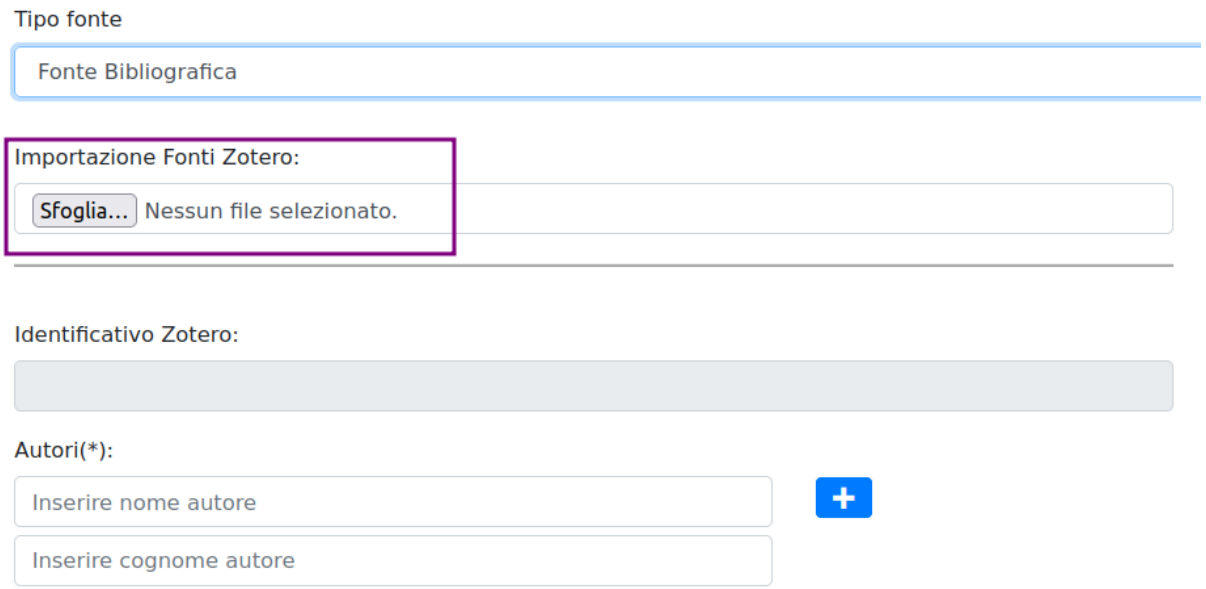

Figura 5.16: Porzione della scheda Fonte Bibliografica

due possibilità: inserire manualmente una Fonte Bibliografica, come già visto nel caso d'uso precedente, o importare un file che ne contiene un certo numero ad affidare al sistema l'analisi automatica di ciascuna di esse. Il risultato di entrambe queste possibilità deve essere il medesimo: quale che sia il metodo scelto, le Fonti avranno la stessa struttura a livello di documenti nel database. Si è svolto dunque un lavoro di mappatura tra i campi definiti da Zotero e quelli staticamente configurati delle Fonti Bibliografiche. Non tutti i campi di Zotero hanno corrispondenza in Fonti Bibliografiche e vice versa. Per essere importata nel sistema, una fonte Zotero deve necessariamente contenere i seguenti campi: Autori, Titolo, Casa Editrice, Data di Pubblicazione e Luogo di Pubblicazione. Se anche solo uno di questi campi non è presente, la Fonte viene scartata dal sistema.

Il tasto ***Sfoglia*** consente di selezionare dal file system dell'utente il documento JSON. Una volta caricato nel browser una finestra modale chiede all'utente se vuole sovrascrivere eventuali duplicati. Dopo aver ricevuto l'istruzione dall'utente il file viene analizzato e trasformato secondo le regole di mappatura stabilite. Questa operazione è svolta nel frontend per evitare di sovraccaricare il server ed evitare attacchi. Il risultato finale è spedito tramite una richiesta AJAX al backend, che effettuerà controlli Fonte per Fonte e si occuperà del loro eventuale salvataggio. Infine, un messaggio contenente un riassunto delle operazioni sarà inviato come risposta al front end.

La figura 5.17 mostra il risultato dell'importazione di sei fonti. Quattro di esse presentano tutti i campi obbligatori e sono correttamente inserite nel sistema. Una non è conforme alle

specifiche e viene dunque scartata. Il suo ID viene mostrato nella finestra di alert per aiutare l'utente ad identificarla nel documento. Un'altra ancora è duplicata e ne viene mostrato il titolo. Una Fonte è contrassegnata come duplicata se ha lo stesso titolo e almeno un autore in comune con una Fonte già esistente nel sistema.

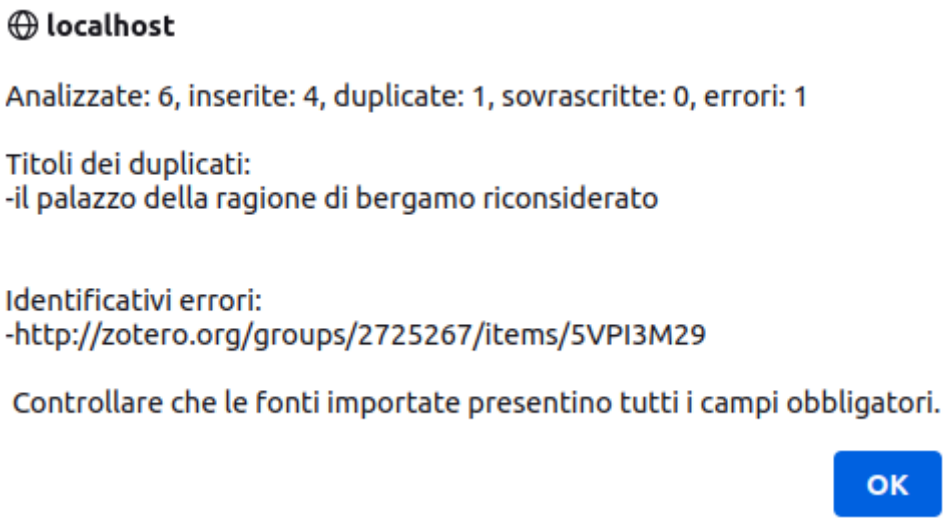

Figura 5.17: Riassunto dell'importazione di più Fonti

Conclusa l'importazione le Fonti correttamente inserite sono visualizzabili nella tabella in figura 5.18. Da qui è possibile accedere a ogni singola Fonte e modificarla. In questo modo è possibile integrare eventuali campi non presenti in Zotero ma di interesse per i ricercatori che utilizzano l'Atlante Digitale.

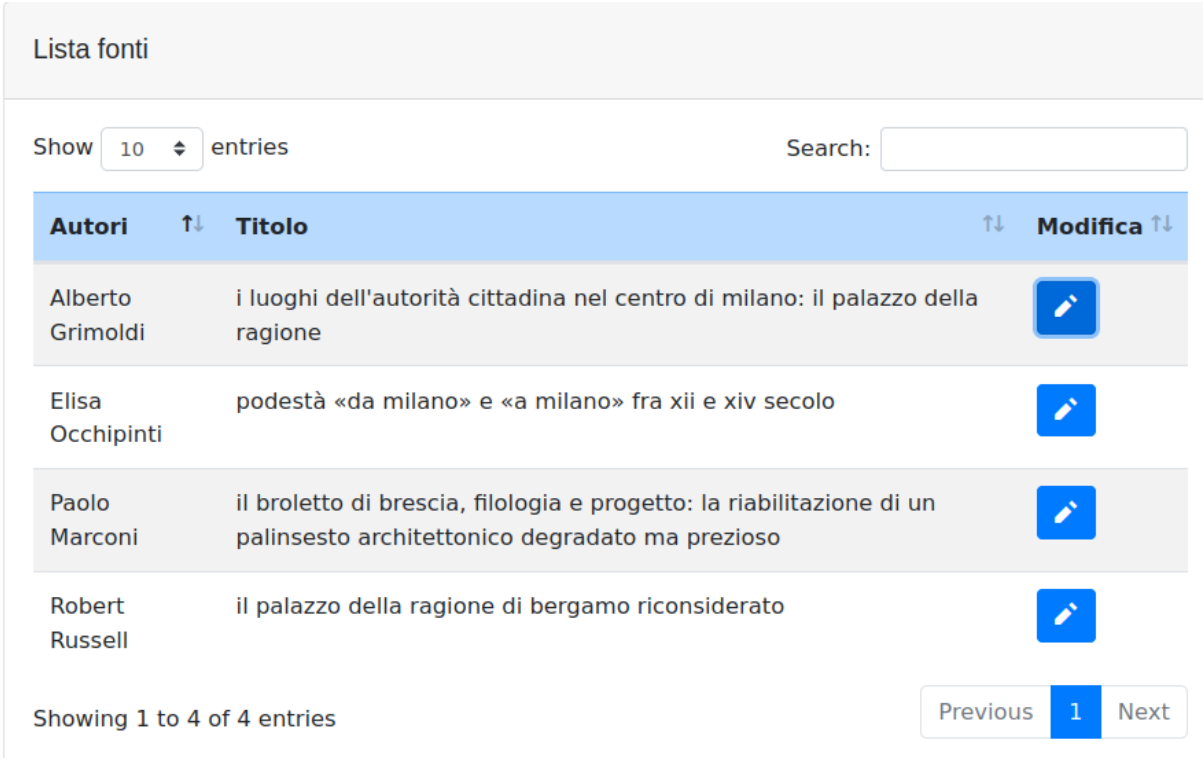

Figura 5.18: Scheda Fonti dopo l'importazione (solo tabella)

5.7 Inserimento Elemento Descrittivo

A questo punto si dispone di tutti gli elementi necessari alla creazione di un Elemento Descrittivo. Il sito prevede tre modi per giungere alla pagina relativa agli Elementi Descrittivi: 1) Dalla barra di navigazione, scheda ***Inserimento di elementi descrittivi*** 2) Dalla scheda di una singola fonte, che presenta il pulsante ***Aggiungi Nuovo Elemento*** oppure 3) All'atto della creazione di una nuova Fonte, quando un pop-up chiede all'utente se si vuole inserire subito un nuovo Elemento Descrittivo. Quale che sia la scelta si viene ricondotti alla pagina mostrata in figura 5.19. L'unica differenza tra i tre percorsi sta nel fatto che il campo Fonti risulta già compilato con la Fonte di riferimento nei casi 2) e 3).

I campi Fonti, Località e Palazzo sono tutti obbligatori e implementati con il plugin **typeahead**. Typeahead trasforma un campo in un menù a tendina le cui opzioni sono le entità già presenti nel database del sistema. Non si può dunque creare un Elemento Descrittivo relativo a un Palazzo, una Fonte o una Località non esistenti. È possibile specificare più Fonti per lo stesso Elemento Descrittivo. Selezionate queste entità bisogna scegliere il Tipo di Elemento Descrittivo, la creazione del quale è descritta dal caso d'uso 5.4. Una volta selezionata la tipologia dell'elemento viene mostrato il form (figura 5.20) che permette di compilare i campi dell'Elemento Descrittivo (figura 5.20). Al salvataggio si viene ricondotti sulla scheda degli Elementi Descrittivi, mostrata in figura 5.21, che presenta tutti gli Elementi creati e presenti nel database.

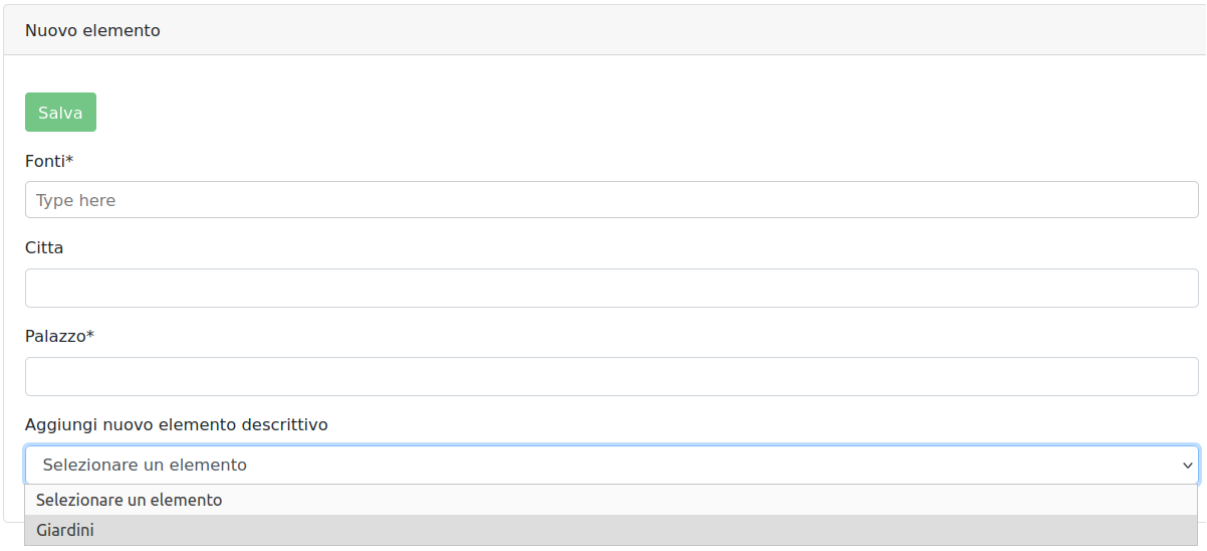

Figura 5.19: Scheda Elemento Descrittivo

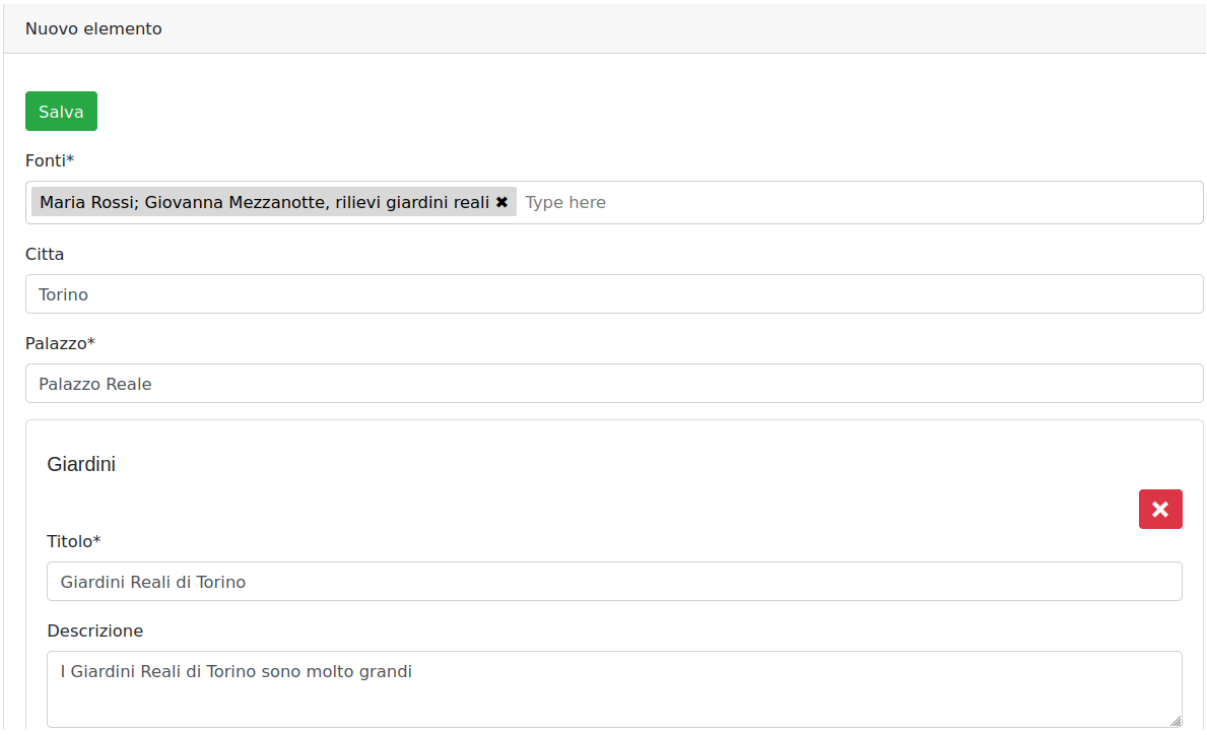

Figura 5.20: Creazione Elemento Descrittivo

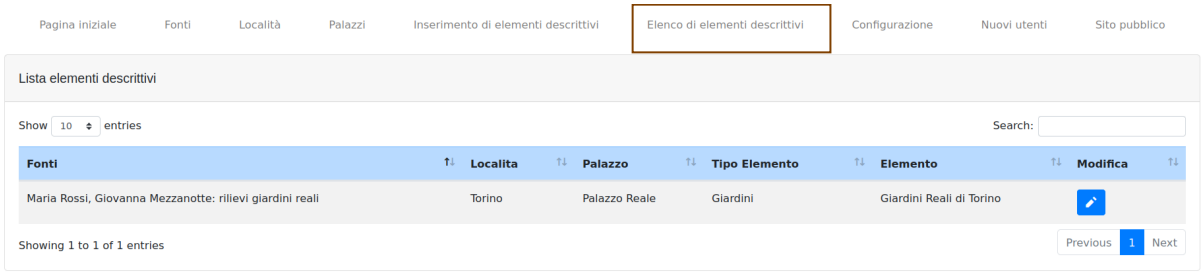

Figura 5.21: Lista Elementi Descrittivi

Capitolo 6

Conclusioni e sviluppi futuri

La tutela del patrimonio artistico e culturale di una nazione ci interessa sotto numerosi punti di vista: definisce parte della nostra identità storica, costituisce materiale di studio per professionisti, contribuisce alla funzione economica di una nazione tramite il turismo ma soprattutto ci qualifica in quanto società. Non riuscire a implementare strategie di prevenzione del rischio, non conservare la memoria storica, permettere che tempo, disastri ambientali e mancanza di interesse rovinino testimonianze preziose del nostro passato è sintomatico del grado di civiltà e maturità raggiunto da una società.

L'Atlante Digitale è stato creato per assolvere principalmente alla funzione di piattaforma di studio e ricerca per professionisti del settore ma il processo di sviluppo sta ampliando le sue potenzialità: la scheda pubblica, sebbene ancora limitata rispetto alla privata, rappresenta un tentativo di coinvolgere il pubblico nella fruizione dei contenuti della piattaforma.

A livello di requisiti funzionali, il sistema presenta sia una componente prettamente visiva, che permette la visualizzazione e geolocalizzazione delle entità di interesse mediante mappe interattive, marker colorati e pop-up sia quella più strettamente legata al lavoro di studio e ricerca. Fonti ed Elementi Descrittivi, tasselli fondamentali del Data Model, costituiscono una delle peculiarità maggiori dell'Atlante. Nel Capitolo 2 si è descritta una serie di sistemi simili al nostro per funzionalità o ambito di studio. Per quanto questi risultino all'avanguardia riguardo a determinati aspetti, come ad esempio la disponibilità di immagini a 360 gradi, virtual tour e altro, nessuno permette una gestione del materiale testuale così come il nostro sistema. La configurazione dinamica degli Elementi Descrittivi, strumento messo a disposizione degli end-user del sistema e che non richiede interazione diretta con gli sviluppatori, apre un ventaglio di possibilità che un insieme di entità definite a priori e staticamente non può garantire.

Il sistema presenta poi numerosi strumenti che lo rendono versatile e flessibile: è possibile compilare una ricca scheda descrittiva per ogni Palazzo, accessibile anche al pubblico; l'importazione di fonti da Zotero è un esempio di integrazione con altri strumenti usati nell'ambito di studio e ricerca; il caricamento di documenti in vari formati rende più completo il lavoro sulle fonti bibliografiche; il pannello amministrativo consente un'agile gestione degli utenti con diversi livelli di privilegio; la disponibilità di due lingue (Inglese e Francese) sia per quanto concerne la navigazione del sito sia per la stesura delle delle descrittive dei palazzi estende la fruibilità del sistema.

Nonostante già ora l'Atlante Digitale sia utilizzabile e utilizzato per svolgere il compito per cui è stato creato, è possibile pensare a una serie di funzionalità che potrebbero interessare sviluppi futuri. A ora Palazzi, Fonti ed Elementi Descrittivi sono entità logicamente legate tra esse ma non ancora integrate a livello di visualizzazione: una scheda riassuntiva del Palazzo, riportante tutto il materiale ad esso collegato secondo le relazioni definite dal data model, potrebbe costituire un miglioramento della user experience, in quanto eviterebbe all'utente di navigare nel sistema per ricercare informazioni che ha senso riunire in un'unica presentazione. Anche uno strumento di ricerca più raffinato e dettagliato, magari con la presenza di filtri, potrebbe migliorare la gestione delle entità e soprattutto delle fonti bibliografiche.

Interessante sarebbe anche introdurre il concetto di "tempo" all'interno dell'Atlante: sempre facendo riferimento ai sistemi presentati nel Capitolo 2, è interessante la presenza di feature che consentono di visualizzare immagini e documenti ordinati temporalmente. La costruzione di un'iconografia storica potrebbe infatti potenziare la componente visiva dell'Atlante, così come l'aggiunta di immagini a 360 gradi o modelli 3D.

A livello di interazione con il pubblico non sarebbe sensato rendere collaborativa la piattaforma, poiché l'ambito di studio è piuttosto specifico e dunque è improbabile che possa arrivare un contributo da parte di un utente esterno. Potrebbe però essere interessante sviluppare una sezione commenti, anche solo per ricevere feedback riguardo al funzionamento del sistema.

Considerando quanto fatto fino a ora e quanto verrà aggiunto e migliorato in futuro, ci sono le premesse per dire che "l'Atlante Digitale dei palazzi comunali" ha le potenzialità per essere una piattaforma importante nell'ambito della ricerca storica e della tutela del nostro patrimonio culturale.

Bibliografia e Sitografia

[1] *HistAntArtSI*. URL: http://www.histantartsi.eu/project.php.

[2] *The Medieval Kingdom of Sicily*. URL: https://kos.aahvs.duke.edu/.

[3] *Mapping Gothic France*. URL: https://mcid.mcah.columbia.edu/art-atlas/mapping-gothic/.

[4] *Mapping Gothic France: About*. URL: https://mcid.mcah.columbia.edu/node/97776.

[5] Giuseppina Vacca. *A Spatial Information System (SIS) for the Architectural and Cultural Heritage of Sardinia (Italy)*. URL: https://www.mdpi.com/2220-9964/7/2/49/htm.

[6] *Carta del Rischio*. URL: http://www.cartadelrischio.beniculturali.it/.

[7] *THEMAS - Thesaurus Management System*. URL: https://www.ics.forth.gr/isl/themas-thesaurus-management-system.

[8] *ResCult - Increasing Resilience of Cultural heritage*. URL: https://www.rescult-project.eu/.

[9] *Hermoupolis Digital Heritage Management (HERMES)*. URL: https://hermoupolis.omeka.net/.

[10] *Arches - An open source data management platform for the heritage field*. URL: https://www.archesproject.org/.

[11] *Information System*. URL: https://en.wikipedia.org/wiki/Information_system.

[12] *Containers vs. Virtual Machines (VMs): What's the Difference?* URL: https://www.ibm.com/cloud/blog/containers-vs-vms.

[13] *Docker*. URL: https://en.wikipedia.org/wiki/Docker_(software).

[14] *Containers vs. virtual machines*. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm.

[15] *Docker overview*. URL: https://docs.docker.com/get-started/overview/.

[16] URL: Nordicapis.com.

[17] *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/.

[18] *Use bind mounts*. URL: https://docs.docker.com/storage/bind-mounts/.

[19] *Use volumes*. URL: https://docs.docker.com/storage/volumes/.

[20] *What's the difference between Docker Compose and Kubernetes?* URL: https://stackoverflow.com/questions/47536536/whats-the-difference-between-docker-compose-and-kubernetes.

[21] *Overview of Docker Compose*. URL: https://docs.docker.com/compose/.

[22] *NoSQL vs SQL Databases*. URL: https://www.mongodb.com/nosql-explained/nosql-vs-sql.

[23] *Bson specification*. URL: https://bsonspec.org/.

[24] *Geospatial Queries*. URL: https://docs.mongodb.com/manual/geospatial-queries/#std-label-geospatial-geojson.

[25] *Aggregation*. URL: https://docs.mongodb.com/manual/aggregation/.

[26] *Replication*. URL: https://docs.mongodb.com/manual/replication/.

[27] *Transactions*. URL: https://docs.mongodb.com/manual/core/transactions/.

[28] *eb*.

[29] *Front-end Frameworks*. URL: https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/.

[30] *jquery*. URL: https://it.wikipedia.org/wiki/JQuery.

[31] *React*. URL: https://it.wikipedia.org/wiki/React_(web_framework).

[32] *React Pros and Cons: What are the Advantages and Disadvantages of ReactJS?* URL: https://www.koombea.com/blog/react-pros-and-cons-what-are-the-advantages-and-disadvantages-of-reactjs/.

[33] *The Progressive Framework*. URL: http://slides.com/evanyou/progressive-javascript#/20/0/7.

[34] *Vue: Guide*. URL: https://vuejs.org/v2/guide/.

[35] *Vue: Comparison with other frameworks*. URL: https://vuejs.org/v2/guide/comparison.html.

[36] *Vue vs React: Comparison of Best JavaScript Frameworks*. URL: https://www.monocubed.com/vue-vs-react/.

[37] *how$_i$t$_w$orks*. URL: https://www.educba.com/how-angular-works/.

[38] *Angular: Pros and Cons*. URL: https://dev.to/siddharthshyniben/angular-pros-and-cons-m9l.

[39] *Bootstrap: The Most Popular HTML, CSS and JS library in the world*. URL: https://getbootstrap.com/.

[40] *Model View Controller*. URL: https://en.wikipedia.org/wiki/Model-view-controller.

[41] *Ruby on Rails*.

[42] *Ruby: introduction*.

[43] *Django*. URL: https://docs.djangoproject.com/.

[44] *Python*. URL: https://it.wikipedia.org/wiki/Python.

[45] *Object Relational Mapping*. URL: https://it.wikipedia.org/wiki/Object-relational_mapping.

[46] *RestAPIS*. URL: https://www.ibm.com/cloud/learn/rest-apis.

[47] *Data Model*. URL: https://en.wikipedia.org/wiki/Data_model.

[48] *Zotero*. URL: https://www.zotero.org/.