



UNIVERSITÀ DEGLI STUDI DI FIRENZE
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria Informatica

**METODOLOGIE DI MIGRAZIONE DA
ARCHITETTURE DESKTOP AD ARCHITETTURE
WEB LIGHTWEIGHT IN ECOSISTEMA JAVA**

Candidato
Fabio Luccioletti

Relatore
Prof. Enrico Vicario

Correlatore
Ing. Boris Brizzi

Anno Accademico 2020-2021

Indice

1	Stato dell'arte	1
1.1	Tecnologie di partenza	4
1.2	Da desktop application a web application	5
2	Metodologia	7
2.1	Architettura di partenza	7
2.2	Architettura di arrivo	10
3	Principi generali del modello di transizione	12
3.1	Dalla view al service layer	12
3.2	Business logic	13
3.3	Data layer	14
4	Caso di studio	17
4.1	Desktop Application	18
4.2	Web Application	21
5	Frameworks	26
5.1	Java Spark	26
5.1.1	Introduzione al framework	26
5.1.2	Elementi di complessità	28

5.2	Ebean	28
5.2.1	Introduzione al framework	29
5.2.2	Elementi di complessità	30
5.3	Elasticsearch	31
6	Prospettiva di progettazione	33
7	Prospettiva di implementazione	37
7.1	Implementazione del Service Layer	37
7.1.1	Web Application	37
7.2	Implementazione del Controller	39
7.2.1	Desktop Application	39
7.2.2	Web Application	39
7.3	Implementazione del Gateway	42
7.3.1	Desktop Application	42
7.3.2	Web Application	44
8	Conclusioni	45
8.1	Utilizzo dei Data Transfer Object	45
8.2	Analisi delle prestazioni	47
	Bibliografia	50

Capitolo 1

Stato dell'arte

Nel procedimento di migrazione da **desktop application** a **web application**, il primo passo è quello di identificare un'architettura *target*, sulla quale modellare l'applicazione.

La scelta dell'architettura rappresenta un passaggio fondamentale; l'adozione di un modello rispetto ad un altro può portare in un caso alla semplificazione dello sviluppo, alla riusabilità dei vari componenti, ad una veloce comprensione del codice e alla scalabilità del progetto, mentre nell'altro alla difficoltà a rispettare ognuno di questi requisiti.

Secondo questa analisi, un modello logico appropriato per la finalità, in grado di aiutare lo sviluppatore nella successiva realizzazione del software è l'**architettura orientata ai servizi** (*SOA*) ed in particolare quella a *tre livelli*. [1]

Una divisione in livelli (o *layer*) è naturalmente associata ad una separazione logica del codice ed è infatti strettamente collegata al concetto di *modularità*: ogni modulo rappresenta una porzione del programma principale, raggruppa metodi appartenenti allo stesso ambito ed è facilmente analizzabile (e testabile) indipendentemente dagli altri moduli.

Scendendo più nel dettaglio di un'architettura orientata ai servizi, i 3 livelli che normalmente possiamo identificare sono il **presentation layer**, l'**application layer** ed il **data layer**:

- il *presentation layer* è il livello che direttamente si interfaccia con gli utenti, fornendo loro le informazioni di interesse e più in generale il servizio per il quale l'applicazione è stata progettata.

Può implementare un'interfaccia grafica per la presentazione dei dati; nel caso questa non sia richiesta, per esempio in una *REST application*, è più appropriato il termine **service layer**.

- il *business-logic layer* incorpora la **logica di programma** e rappresenta pertanto il cuore del progetto, che fa da intermediario tra l'interfaccia utente e i dati grezzi dei quali si dispone. Nasconde pertanto

la complessità delle operazioni al livello superiore, permettendogli al contempo di utilizzare le proprie funzioni esponendo delle *primitive*.

- il *data layer*, si occupa della lettura, scrittura e gestione dei dati, immagazzinati in un *database*.

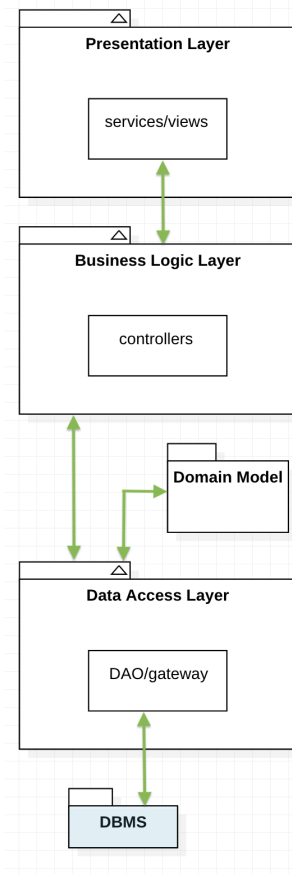


Figura 1.1: Service Oriented Architecture

Fornire un servizio dividendo la logica del programma in più livelli è una scelta conveniente per più di un motivo:

- **Indipendenza:** ogni livello può essere sviluppato in maniera indipendente dagli altri, eventualmente anche da persone differenti all'interno del team. Infatti un livello espone dei metodi che possono essere chiamati dal layer superiore, ma la logica di funzionamento dei metodi è trasparente al chiamante, che è interessato solo al valore di ritorno. La divisione dei livelli, tipica di applicazioni modulari, permette una migliore specializzazione e quindi di lavorare seguendo i principi di **single responsibility** e **separation of concerns**: un programma in cui ogni componente è responsabile di un set limitato di azioni, appartenenti ad uno stesso contesto, oltre ad essere obbiettivo naturale (si pensi ad esempio all'efficienza delle catene di montaggio), aiuta a non duplicare il codice (*DRY - do not repeat yourself*) ed è più facile da mantenere.
- **Scalabilità:** diretta conseguenza dell'indipendenza è la scalabilità; ogni livello può essere ampliato o distribuito su più server nel caso fosse necessario gestire un alto numero di richieste. In ogni caso non è richiesto che su ogni server siano presenti tutti i layer.
- **Affidabilità:** Sviluppare e mantenere progetti di dimensione ridotta è tendenzialmente più semplice che gestire applicazioni di grandi dimensioni; la modularità intrinseca dell'architettura orientata ai servizi permette di testare e fare il *debug* del codice con più facilità, e quindi di avere un servizio più affidabile.
- **Efficienza di sviluppo:** ogni livello raggruppa metodi che si riferiscono ad uno stesso contesto e che possono essere curati da sviluppatori

specializzati nel settore: il *web designer* si può dedicare al livello presentazione, il *software engineer* alla logica di programma ed il *database administrator* può occuparsi del livello dati.

1.1 Tecnologie di partenza

Una desktop application sviluppata in ecosistema Java implementa generalmente una serie di tecnologie e *framework* di supporto alla realizzazione del software.

Nella sua forma più semplice, un'applicazione completa deve poter relazionarsi con una struttura dati e presentare i risultati all'utente: partendo dal livello più alto nello stack (presentation layer), un framework molto utilizzato per lo sviluppo di interfacce grafiche è **Swing**.

Swing rappresenta una soluzione semplice per la creazione di *GUI* in Java, basata su componenti gestiti direttamente dalla *Java Virtual Machine* e pertanto indipendenti dalla piattaforma. Una soluzione del genere non è assolutamente comparabile, in quanto a versatilità e personalizzazione, con la coppia *HTML + CSS* che caratterizza il web: il numero di *widget* presenti sono limitati (anche se nuove classi possono essere aggiunte integrando la libreria *open source* **SwingX**) e l'interfaccia grafica presenta uno stile standardizzato in tutte le applicazioni.

Spostandosi invece sul fondo dello stack (data layer), la comunicazione standard con il *database* viene gestita attraverso il connettore **JDBC** (Java DataBase Connectivity). In realtà, anche se mascherato, questo driver è spesso presente in tutte le applicazioni, anche quelle che utilizzano framework di *persistenza* come *Hibernate* od *Ebean* (descritto in maniera più approfondita

dita più avanti nella trattazione).

Per quanto riguarda la base dati vera e propria, le scelte tecnologiche possono essere differenti, anche in base alle esigenze di sviluppo. Nel caso di studio che verrà presentato in seguito, il **DBMS** è **PostgreSQL**, un *database management system* **relazionale**, open-source e adeguatamente avanzato: implementa ad esempio nativamente la ricerca **full-text**, per la ricerca di un documento in base a delle parole chiave.

1.2 Da desktop application a web application

Quando l'applicazione desktop (*standalone* ed installata direttamente sul dispositivo dell'utente) diventa una limitazione per il corretto utilizzo del software, magari per la necessità di download e installazione degli aggiornamenti oppure per l'impossibilità di utilizzare il servizio su piattaforme differenti (come anche un dispositivo mobile), una soluzione può essere la distribuzione del software sulla rete.

È importante specificare che non necessariamente la web application rappresenta la naturale evoluzione della desktop application. La decisione di procedere con la transizione deve seguire un'attenta analisi dei requisiti e la valutazione dei vantaggi e degli svantaggi in relazione alle caratteristiche del servizio offerto.

In generale possiamo evidenziare alcuni evidenti benefici per l'utente derivanti dall'utilizzo di un'applicazione web:

- non è necessario **installare** il software; tutti i servizi sono accessibili dal browser ed ospitati su server remoti. Per lo stesso motivo gli aggiornamenti non sono una responsabilità dell'utente e spesso possono

essere installati senza condizionare in maniera significativa l'esperienza dell'utilizzatore.

- i servizi sono accessibili da **qualsiasi piattaforma**, anche da ambiente mobile, fintanto che è disponibile una *connessione internet*. Dato che la computazione avviene sul server che ospita il servizio, la **potenza di calcolo** del dispositivo dell'utente passa in secondo piano: un software, se *responsivo* è accessibile anche da dispositivi mobili.

La stessa analisi dei requisiti permette poi di definire quali tecnologie possono essere adottate per lo sviluppo dell'applicazione web. Nel caso di progetti di taglia medio/piccola, come vedremo nel caso di studio presentato in seguito, può essere appropriato utilizzare tecnologie leggere, che permettano un approccio agile e efficiente.

Procedendo in questa direzione, una delle possibili soluzioni è rappresentata dall'utilizzo del *framework* **Spark** al livello presentazione e di **Ebean** a livello dati, entrambe tecnologie lightweight che verranno approfondite in un capitolo successivo.

Capitolo 2

Metodologia

Dopo aver studiato i vantaggi di un'architettura orientata ai servizi, è necessario stabilire il metodo a supporto della transizione da un'applicazione sviluppata con *Java Standard Edition* per un'ambiente desktop ad un equivalente applicativo web.

Definiamo con maggiore precisione il modello di partenza e quello di arrivo.

2.1 Architettura di partenza

L'applicativo di partenza è stato progettato già nell'ottica della transizione, pertanto emula a grandi linee la struttura target della web application. Si tratta di una Desktop Application realizzata con Java (Standard Edition). L'architettura, ben ingegnerizzata, mantiene un certo grado di libertà grazie all'utilizzo di due *pattern architetturali*:

- **MVC**

L'applicazione è dotata di un'interfaccia grafica, realizzata con java *Swing* e *Swingx*. Pertanto è stato adottato il pattern MVC (**model-view-controller**), per la progettazione e strutturazione di applicazioni

di tipo interattivo. L'applicazione risulta divisa in tre parti (corrispondenti nel codice agli omonimi package): il **model** che rappresenta il modello dei dati di interesse per l'applicazione, il **controller** che definisce la logica di controllo e le funzionalità applicative, la **view** che gestisce la logica di presentazione dei dati all'utente.

Il pattern MVC opera in maniera molto semplice: la view rappresenta le pagine che vengono mostrate all'utente e con cui questo può interagire; ad ogni azione dell'utente viene invocato un metodo del controller, che a sua volta chiama e *gestisce* un metodo di un **gateway**. Quando le operazioni (di modifica o richiesta dati) sono concluse, il controller ritorna un valore, o semplicemente il controllo, alla view che lo aveva invocato.

- **Gateway (DAO)**

Il dialogo con una base dati è un aspetto importante dell'applicazione; i *controller* e le altre classi facenti parte dell'**application logic** hanno spesso necessità di ricevere specifiche informazioni dal **data layer**. Per nascondere la complessità di quest'ultimo al resto dell'applicazione è stato usato il pattern **Data Access Object** (o **Table Data Gateway**, nel lessico di *Martin Fowler* [2]). Ognuno dei Gateway incapsula una tabella del database (talvolta più di una se il loro contesto di utilizzo è il medesimo) e contiene nei suoi metodi tutto il codice *SQL* necessario all'interrogazione.

Il **Model Diagram** in figura sintetizza l'architettura di partenza: è evidente una prima divisione in livelli, che troveranno una corrispondenza nel prodotto finito, spesso tramite una semplice ristrutturazione e non una riscrittura da zero del codice.

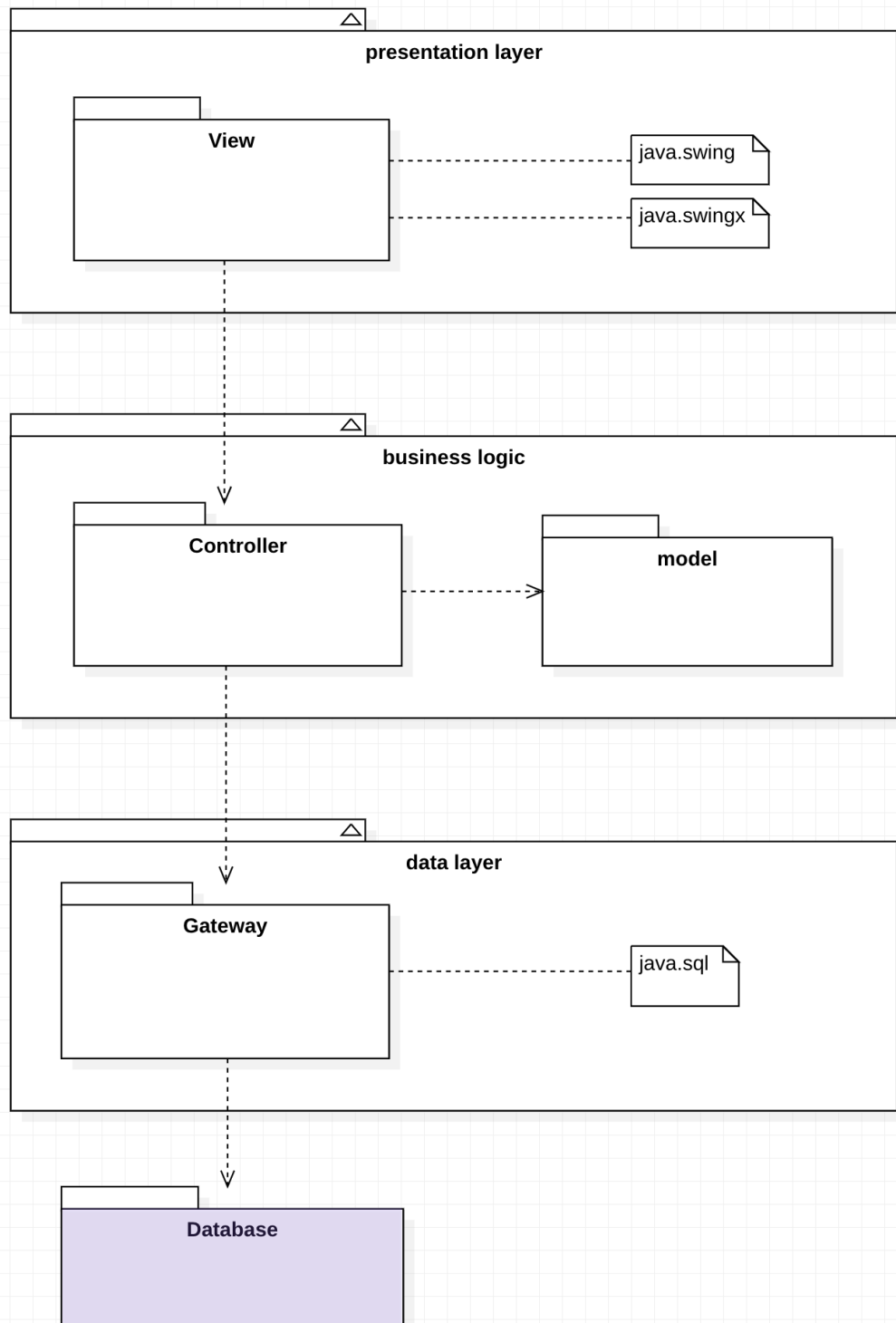


Figura 2.1: Model Diagram per l'architettura di partenza

2.2 Architettura di arrivo

Il Model Diagram di arrivo, come anticipato, mantiene la stessa struttura a livelli dell'applicazione di partenza, ma rivista nel paradigma dell'architettura orientata ai servizi. Integra però framework per la gestione della comunicazione tramite HTTP (*Spark*) e per la gestione ed interrogazione del database (*Ebean*).

È importante sottolineare come in questo caso il **front-end** rappresenti un'applicazione a sé stante, che si relaziona con il service layer.

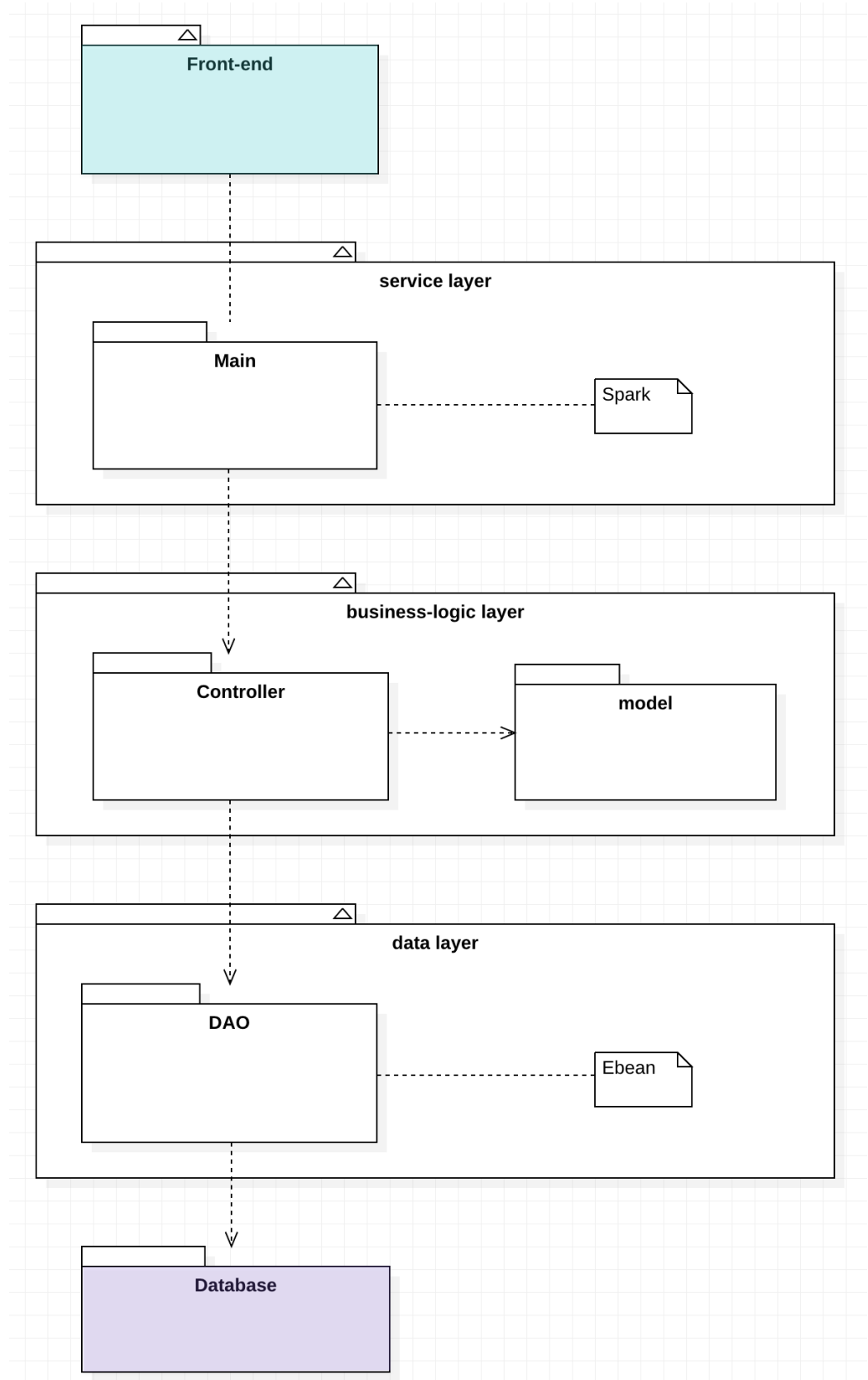


Figura 2.2: Model Diagram per l'architettura di arrivo

Capitolo 3

Principi generali del modello di transizione

Per una prima formalizzazione della metodologia di transizione da desktop application a web application, possiamo descrivere in maniera generale la transizione, astruendo tecnologie e casi d'uso particolari.

Come abbiamo visto, esistono già similitudini a livello strutturale tra l'architettura di partenza e quella di arrivo, in particolare per quanto riguarda l'organizzazione a livelli: la trattazione di questo capitolo cercherà perciò di mantenere questa organizzazione, soffermandosi separatamente sulla descrizione di ogni livello, a partire da quello più alto.

3.1 Dalla view al service layer

Il livello che direttamente si interfaccia con l'utente subisce una sostanziale trasformazione nel passaggio da desktop a web application, non soltanto per l'utilizzo di tecnologie differenti (come *web framework* per la gestione degli *endpoint*), ma soprattutto a causa di un profondo cambio di **paradigma**:

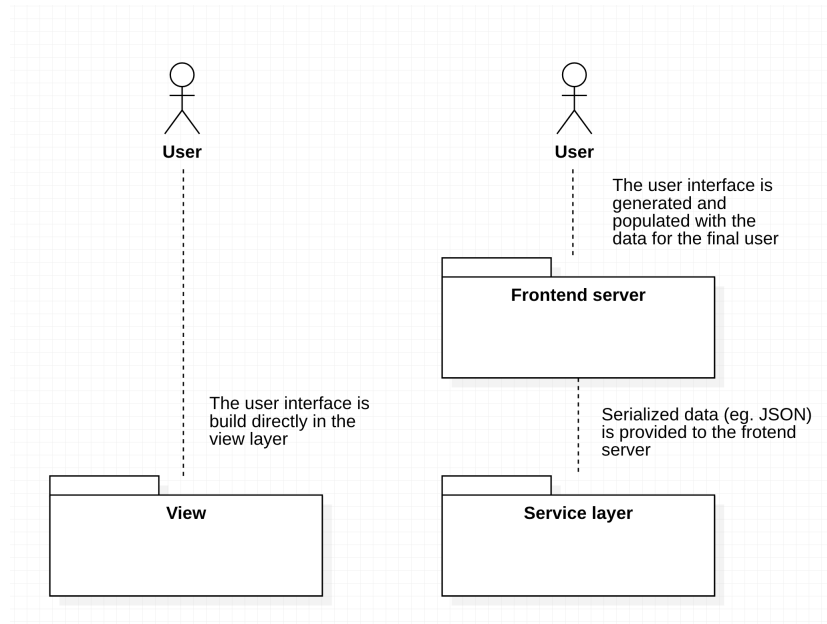


Figura 3.1: View to Service Layer

non sono più esposte **viste** ma **servizi**.

La *View* della desktop application non soltanto fornisce le informazioni di interesse all'utente, ma rappresenta lo stesso *front-end* e deve pertanto implementare l'interfaccia grafica. Di contro, il *service layer* di una applicazione web si relaziona con l'utente, ma *attraverso* un front-end, a cui deve solamente fornire i dati serializzati demandando l'implementazione grafica.

3.2 Business logic

Spostandosi al livello inferiore arriviamo alla business logic (package Controller per la desktop application). Dato che a questo livello è implementata la logica di programma, mantenendo invariate le funzionalità dell'applicazione, le variazioni sono minime: il controller continua a gestire le eccezioni, si relaziona con il gateway ed espone funzionalità per il livello superiore.

Gli unici cambiamenti degni di nota sono dovuti alla sostanziale trasformazione del service layer: non sono più necessarie tutte le funzioni di controllo per interfaccia grafica ma si rendono necessari nuovi strumenti per la gestione della comunicazione web, in particolare:

- la **serializzazione** dei dati, successivamente esposti al client dal livello superiore. Un formato molto comune per la trasmissione dei dati attraverso il protocollo HTTP è **JSON** (*JavaScript Object Notation*): il controller gestisce la trasformazione degli oggetti in formato JSON ¹ e crea il *body* della risposta, successivamente inviata dal livello superiore.
- la scelta dei **codici di stato** della risposta HTTP, aspetto non necessario nel contesto di una desktop application, ma importante per una corretta comunicazione in rete: per esempio, la presenza di una eccezione, che devia il normale flusso di esecuzione, potrebbe portare ad avere dati inconsistenti e inaspettati per il front-end. Se durante la gestione dell'eccezione il codice di stato viene posto a *500* (*Internal Server Error*), è probabile che il front-end sia in grado di interpretare l'errore e di conseguenza gestirlo in maniera adeguata.

3.3 Data layer

Se a livello di logica di programma l'influenza della tipologia di architettura (desktop o web) era relativa, spostandosi al livello più basso dello stack (data layer) gli effetti sono ancora meno evidenti, se non completamente assenti. Nell'ambito della comunicazione con la base dati, non è importante sapere come avviene la trasmissione delle informazioni al client; i risultati

¹nei casi più complessi, la serializzazione può non essere una responsabilità diretta del controller, ma è piuttosto implementata dai *DTO* o dai *Mapper*

delle query di interrogazione al database vengono forniti al controller, che rappresenta un muro tra il data layer e il service layer, nascondendo all'uno le caratteristiche dell'altro.

In questo caso, è la scelta della tecnologia e dei framework ad impattare in maniera *determinante* sulla forma del data layer, ed in particolare dei Gateway.

Un modello di transizione che abbia come obiettivo anche la *semplificazione*, *portabilità* ed *efficienza* del codice normalmente prevede un'interazione con il database non solo attraverso le classiche *query SQL* (o NoSQL) ma anche con la tecnica dell'*Object Relational Mapping (ORM)*. L'utilizzo di un framework ORM prevede, tra le altre cose:

- **l'astrazione** della tecnologia utilizzata dal DBMS; non è quindi più necessario modificare le query in base al tipo di database presente, ma normalmente solo poche righe del file di configurazione. Eventuali *factory* per la creazione di gateway personalizzati in base alla tecnologia del database non sono più necessari.

L'indipendenza dal linguaggio di interrogazione rappresenta in pratica anche un vantaggio per lo sviluppatore Java, che può sfruttare la semantica di un linguaggio familiare senza doversi ricordare le sottili differenze tra, ad esempio, *MySQL*, *PostgreSQL* ed *SQLite*;

- **gestione interna delle connessioni** al database; le connessioni non devono più essere gestite attraverso un *connection pool* implementato dallo sviluppatore, così come gli *statement* ed i *result set* non devono essere esplicitamente rilasciati e chiusi (ad esempio all'interno di blocchi *try - catch*).

Spesso la gestione delle connessioni si traduce anche in una gestione della concorrenza nell'accesso ai dati;

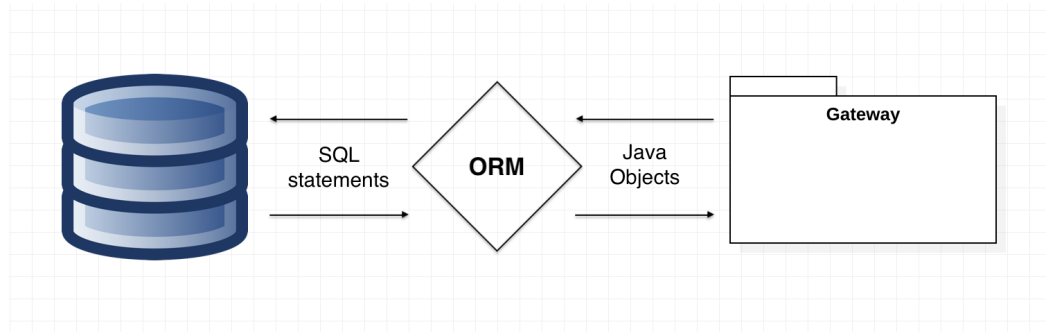


Figura 3.2: ORM mapping

- **mappatura del modello** alle tabelle del database e la possibilità di scrivere query attraverso la concatenazione di metodi (propria delle query ORM), che riduce sensibilmente la quantità di codice sorgente. Lavorare con degli oggetti Java rispetto che con dati più grezzi rappresenta un grande vantaggio in quanto ne facilita la manipolazione successiva: a questo proposito alcuni framework implementano anche metodi di serializzazione automatica (nei formati JSON, XML, ecc), delle entità recuperate dal database.

La semplificazione del data layer è sicuramente uno degli aspetti centrali della migrazione.

Gestire la comunicazione con il database senza l'ausilio di un framework di supporto aumenta significativamente la complessità del codice.

Capitolo 4

Caso di studio

Il caso di studio è un'applicazione software per la gestione e la ricerca di informazioni relative ad un gran numero di musei italiani. Il database di partenza contiene circa 7500 musei le cui informazioni sono state estratte da Wikipedia tramite un processo di **web scraping**.

L'applicazione permette agli utenti di ricercare i musei presenti nel database grazie ad una barra di ricerca, che può essere impostata utilizzando differenti strategie: la ricerca può basarsi solamente su *parole chiave* oppure limitare i risultati per prossimità in base ad una *località* fornita dall'utente.

In generale, le parole chiave inserite dal client vengono utilizzate per una ricerca *full-text*, che restituisce una lista di musei in base all'attinenza (i campi presi in considerazione per calcolare lo *score* di ogni museo sono il nome e la descrizione, che può essere anche molto lunga). Se viene inserita anche una località, questa viene inviata alle **API Nominatim** di *OpenStreetMap*, che la interpretano e restituiscono delle coordinate geografiche, utilizzate successivamente per filtrare i risultati in base alla distanza.

Dai risultati di ricerca, è possibile recuperare tutte le informazioni relative ad un museo di interesse: nome, descrizione, locazione geografica, categoria, sito

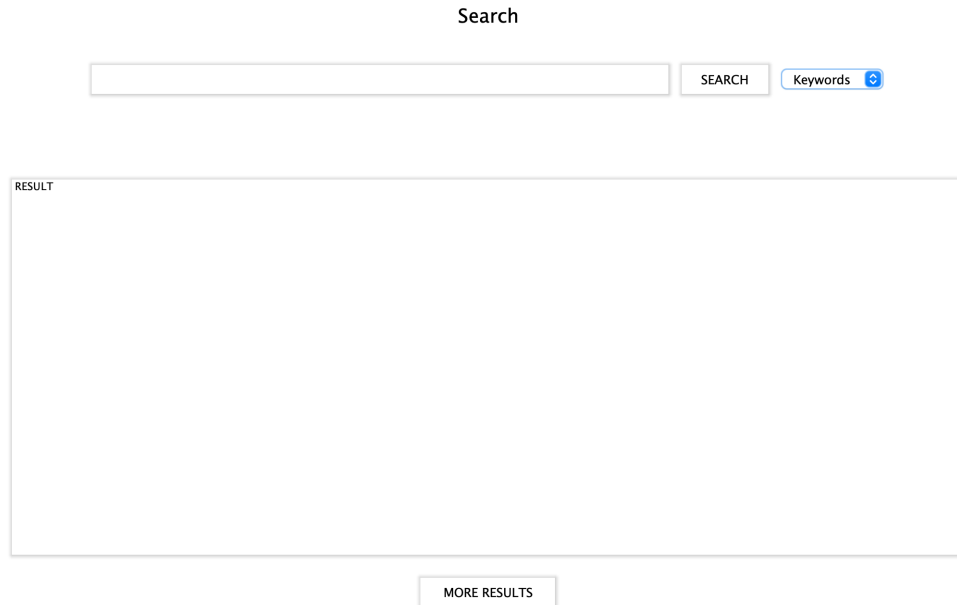


Figura 4.1: Interfaccia di ricerca nella desktop application

web ed un'eventuale immagine. Questi campi possono anche essere modificati da un amministratore (previa autenticazione) ed aggiornati sul database. Sebbene le funzionalità disponibili siano le medesime, la *user experience* necessariamente varia nel passaggio da applicazione desktop ad applicazione web.

4.1 Desktop Application

Nella **applicazione desktop** [4] l'utente interagisce con il software tramite un'interfaccia grafica. Come si può vedere in figura, è presente una barra di ricerca ed un selettore per impostare la strategia desiderata.

La ricerca per località aggiunge un ulteriore campo di testo libero dove l'utente può inserire il luogo di interesse.

I risultati sono presentati in una lista, dove ogni elemento è selezionabi-

Search

SEARCH

Location 


Location

Figura 4.2: Barra di ricerca con strategia basata sulla località

Search

Leonardo da Vinci

SEARCH

Location 

Location Firenze

RESULT

- Galleria degli Uffizi
- Museo nazionale del Bargello
- Museo Galileo
- Casa Vasari (Firenze)
- Basilica di Santa Croce
- Museo dell'Opera di Santa Croce
- Casa Buonarroti
- Villa La Petraia
- Villa medicea di Poggio a Caiano
- Casa natale di Leonardo

MORE RESULTS

Figura 4.3: Interfaccia con risultati di ricerca

le e al click apre la pagina del museo corrispondente. Non tutti i risultati sono restituiti immediatamente, solamente i più pertinenti; eventualmente è possibile visualizzare più elementi premendo il tasto *more results*.

La pagina di un museo presenta all'utente tutte le informazioni disponibili: oltre al nome del museo ed alla descrizione, sempre presenti, è disponibile un collegamento alla pagina Wikipedia del museo ed al sito internet. La schermata è arricchita da un'immagine, da alcune indicazioni sulla locazione del museo e dalle categorie al quale appartiene. In mancanza di uno o più di



Figura 4.4: Pagina del museo

questi campi il layout viene ricalcolato per una migliore esperienza utente.

Il software mette a disposizione per gli amministratori anche una maschera di aggiunta museo di e modifica delle informazioni per i musei esistenti. Nel caso della sola modifica, i campi della maschera sono già popolati con i valori recuperati dal database.

Il sistema prevede anche un processo di login per l'utilizzo del software. L'autenticazione è gestita dal database (PostgreSQL), su cui è installato l'estensione **pgcrypto**, che fornisce una serie di funzioni per il salvataggio criptato delle password inserite dall'utente. Una gestione dei ruoli permette di distinguere tra utente base ed amministratore: il primo ha la possibilità solo di eseguire ricerche e consultare i vari risultati; il secondo ha accesso a funzioni più avanzate come l'aggiunta, modifica e rimozione di un museo.

Galleria degli Uffizi

LOCATION: Piazzale degli Uffizi 6, I-50122 Firenze - Firenze

CATEGORIES: Pinacoteca. Museo di scultura. Museo statale. Museo archeologico

Insert museum data

Name: Galleria degli Uffizi

Wikilink: <https://it.wikipedia.org/?curid=24101>

Website: <https://www.uffizi.it>

Location: Firenze

Latitude: 43.768439

Longitude: 11.2559

Description: La Galleria degli Uffizi è un museo statale di Firenze, che fa parte del complesso museale denominato Gallerie degli Uffizi e comprendente, oltre alla suddetta galleria, il Corridoio Vasariano, le collezioni di Palazzo Pitti e il Giardino dei Boboli, che insieme costituiscono per

Address: Piazzale degli Uffizi 6, I-50122 Firenze

Category: Museo statale

Cancel OK

Figura 4.5: Maschera di modifica delle informazioni

4.2 Web Application

L'applicazione web [3] espone funzionalità all'utente attraverso una **RESTful API**. La comunicazione avviene tramite richieste *HTTP* e le informazioni sono serializzate in formato *JSON*.

In mancanza di un'applicazione di front-end che implementi l'interfaccia grafica, le richieste possono essere inviate da linea di comando tramite **cURL** o con software specifici per il testing come **Insomnia**.

Le risorse sono accessibili tramite **URI** (in questo caso *URL*), eventualmente fornendo dei parametri direttamente nel *path* dell'*URL* oppure nel *body* della richiesta; l'esempio in figura rappresenta una *GET request* per ottenere una lista di musei affini alla query "*Leonardo da Vinci*" e non distanti da *Firenze*. I risultati sono restituiti in un *JSON array* in forma minimale, le eventuali informazioni mancanti possono essere ottenute richiedendo alle API la pagina di un museo tramite il suo id.

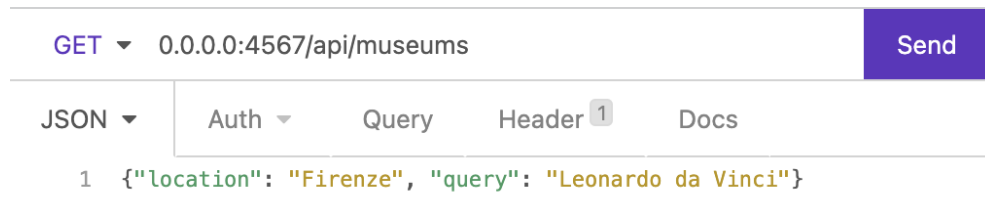
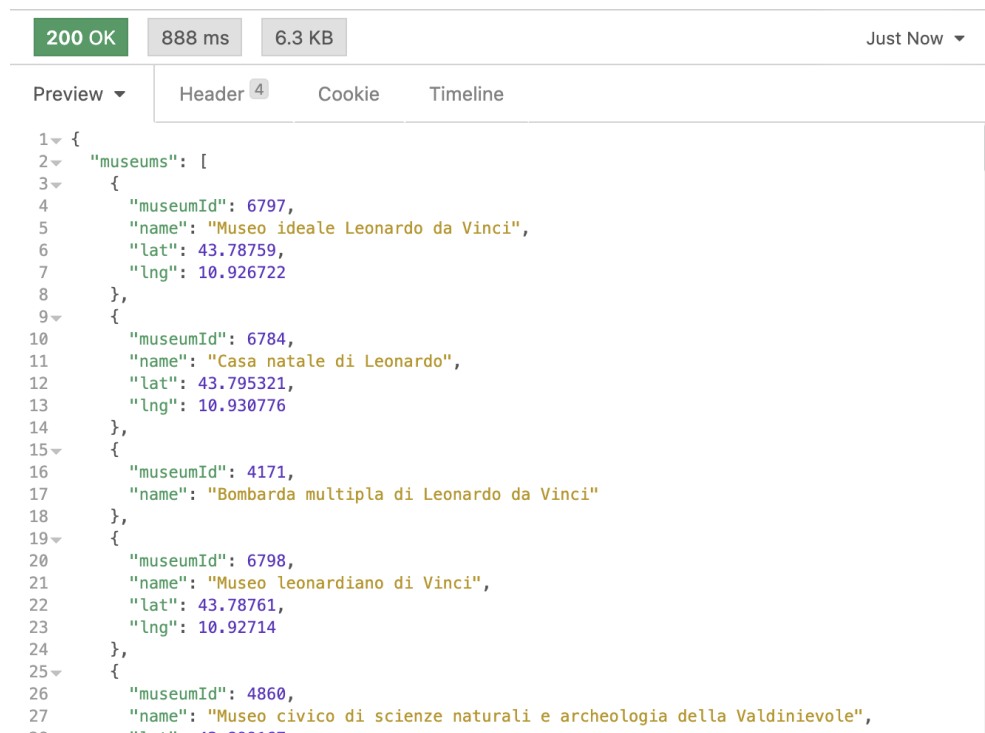
Figura 4.6: Richiesta *GET* per la ricerca dei musei in Insomnia

Figura 4.7: Risultati di ricerca



Figura 4.8: Ricerca tramite id

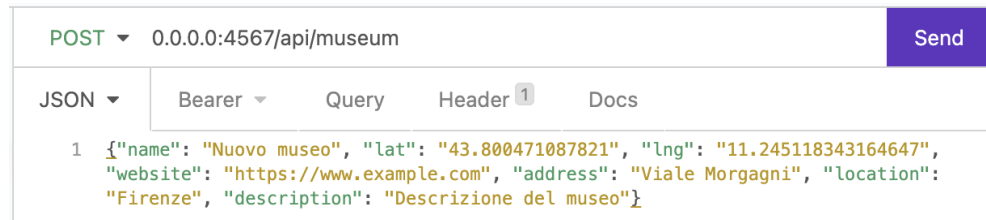


Figura 4.9: Aggiunta di un museo

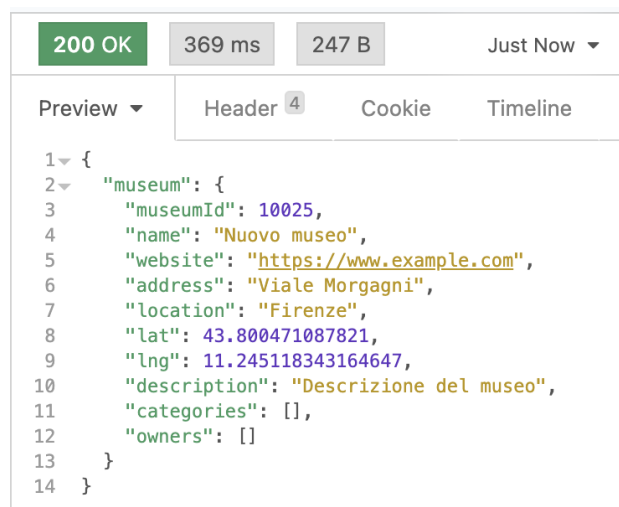


Figura 4.10: Informazioni del nuovo museo

Come nella versione desktop, anche in questo caso l'amministratore¹ ha la possibilità di aggiungere o modificare le informazioni relative ad un museo. La procedura di creazione di un nuovo museo è gestita attraverso una *POST request*, che contiene nel body i parametri da salvare all'interno del database. Se la procedura è andata a buon fine, il software restituisce le informazioni del nuovo museo, compreso il suo id, generato automaticamente.

La modifica del museo mantiene la stessa forma del body, ma è ottenuta tramite una *PUT request*, indicando come parametro del path l'id del museo.

¹La procedura di autenticazione/autorizzazione è gestita dal server tramite la generazione di **token JWT** e dell'implementazione della logica di controllo dei token ricevuti nel campo *Authorization* della richiesta HTTP.

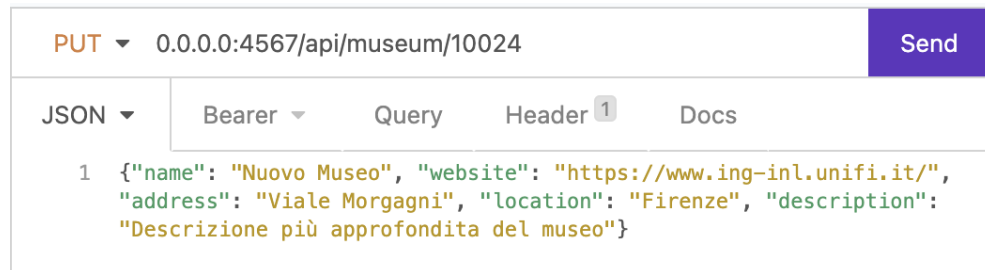


Figura 4.11: Aggiornamento delle informazioni del museo

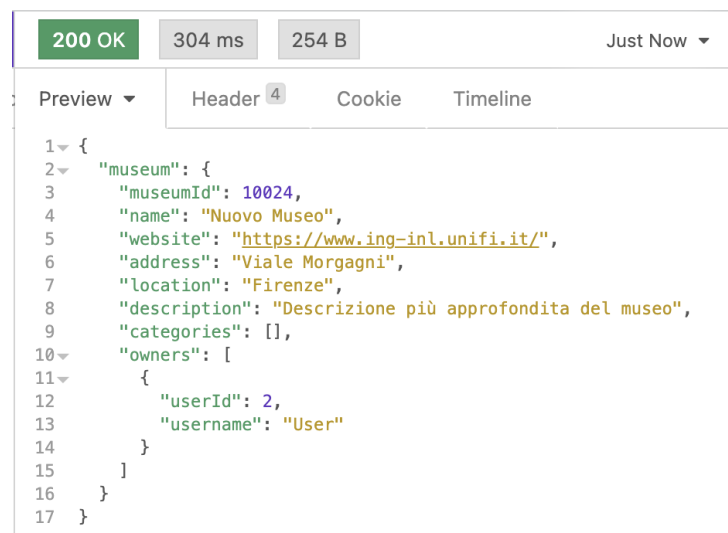


Figura 4.12: Informazioni del museo aggiornate

Anche in questo caso la risposta HTTP contiene nel body tutte le informazioni aggiornate.

Nel caso le operazioni di ricerca, aggiunta o aggiornamento non andassero a buon fine, l'applicativo restituisce un codice di errore con un messaggio personalizzato, in modo che il front-end sia in grado di gestire il problema, in particolare:

- la richiesta o la modifica di un museo che non esiste (l'id non ha corrispondenza nel database) restituisce una risposta con codice di stato **404** (*Not Found*).

- una query di ricerca malformata, la presenza di un parametro non interpretabile (come un id del museo non numerico), l'omissione dei campi obbligatori nella procedura di aggiunta o modifica del museo restituiscono una risposta con codice di stato **422** (*Unprocessable Entity*).
- le eccezioni real-time che non dipendono dalle richieste dell'utente restituiscono una risposta con codice di stato **500** (*Internal Server Error*).

Capitolo 5

Frameworks

Dopo aver elencato i principi generali del modello di transizione possiamo vedere come possano essere applicati al caso d'uso descritto nel capitolo precedente.

Prima di concentrarci sulla progettazione è importante presentare i framework che saranno utilizzati, in modo che possano essere chiari il contesto di utilizzo ed i cambiamenti strutturali dovuti al loro impiego.

5.1 Java Spark

Java Spark è un *web framework* scritto in Java, leggero ed intuitivo che non necessita delle API di *Java Enterprise Edition*.

5.1.1 Introduzione al framework

Spark è un web framework particolarmente indicato per lo sviluppo di applicativi di piccole e medie dimensioni, è in grado di servire pagine



Figura 5.1: Spark logo

web, ma è spesso utilizzato per lo sviluppo di REST API.

Dopo l'installazione delle dipendenze, bastano poche righe di codice per avere un servizio funzionante: mantenendo la configurazione di default, il codice che segue espone una pagina all'indirizzo `http://localhost:4567/hello`.

```
1  import static spark.Spark.*;
2
3  public class HelloWorld {
4      public static void main(String[] args) {
5          get("/hello", (request, response) -> "Hello World");
6      }
7  }
```

Spark lavora nel livello più alto dello stack ovvero nel service layer, fornisce pertanto le sue primitive direttamente all'utente (eventualmente con la mediazione di un front-end). Questo viene fatto esponendo una serie di *endpoint* (URL), definiti all'interno dell'applicazione come *routes*. In realtà gli URL definiscono solo parzialmente una route: è necessario identificare un metodo HTTP (*GET*, *POST*, *PUT*, *DELETE*...) ed una funzione di *callback*.

```
1  get("/museums", (request, response) -> {
2      response.type("application/json");
3      return controller.searchMuseums(request.body());
4  });
```

Quando l'utente interroga il server sul path `/museums` utilizzando il metodo GET, viene eseguita una funzione (lambda), che ha come parametri la richiesta e la risposta. All'interno della funzione non è però direttamente implementata logica di programma, che è demandata al *controller*, ma solo la modifica degli header della risposta.

5.1.2 Elementi di complessità

Spark promette di essere un framework leggero, ma soprattutto semplice. Effettivamente, una conoscenza anche non troppo approfondita di tecnologie simili, come ad esempio *Flask* per *Python*, permette configurare un nuovo progetto in pochissimi minuti.

Non essendo necessarie particolari integrazioni, Spark non mostra evidenti elementi di complessità; tutte le funzionalità sono implementabili seguendo la *documentazione ufficiale*. A titolo di esempio, l'integrazione con il *logger* avviene in maniera automatica. Nel caso di **logback** (come anche *log4j* o più in generale *slf4j*) basta aggiungere la dipendenza al progetto:

```
1 <dependency>
2 <groupId>ch.qos.logback</groupId>
3 <artifactId>logback-classic</artifactId>
4 <version>1.2.3</version>
5 </dependency>
```

5.2 Ebean

Ebean è un framework che fornisce un servizio di *Object-relational mapping* (ORM), ponendosi come intermediario tra la logica di programma ed il *Database* e quindi nel data layer. Rappresenta un'alternativa a piattaforme più pesanti come *Hibernate*.



Figura 5.2: Ebean logo

5.2.1 Introduzione al framework

Ebean rappresenta un'ottima soluzione per lo sviluppatore che è alla ricerca di un compromesso tra *funzionalità offerte*, *facilità di utilizzo* e *risorse utilizzate*.

Tra i DBMS supportati figurano *H2*, *Postgres*, *MySql*, *Oracle* ed *Sqlite* (non sono supportati database non relazionali, come ad esempio *MongoDB*).

Sebbene sia consigliata l'adozione di query ORM, è possibile utilizzare query SQL, che possono essere automaticamente mappate a **Data Transfer Object** (DTO) definiti nel modello.

Ebean gestisce la persistenza dei dati sul database, generando le tabelle e le loro relazioni automaticamente basandosi su un modello (che è definito da oggetti Java ed annotazioni). Una conseguenza è che la scrittura delle query diventa molto più semplice ed efficace:

```
1 /**
2  * Returns a Museum object if the museum with the given id
3   * exists in the database, or <b>null</b> otherwise.
4  *
5  * @param museumId the museum id
6  * @return a Museum object
7  */
8 public Museum getMuseum(long museumId) {
9     return new QMuseum()
10         .museumId.eq(museumId)
11         .categories.fetch(QCategory.alias().name)
12         .owners.fetch(QUser.alias().username)
13         .findOne();
14 }
```

Il metodo descritto, che seleziona un museo in base al suo id, non solo ha una forma molto intuitiva, ma restituisce direttamente un oggetto Mu-

seo dove tutti i campi sono mappati con le relative colonne della tabella nel database; inoltre, se effettivamente richiesto (in base al principio della *lazy initialization*), carica anche eventuali oggetti collegati al museo con relazioni *uno-uno*, *uno-molti*, *molti-molti*.

Sempre nell'esempio è stato deciso di ricavare per il museo anche le sue *categorie* ed i suoi *proprietari*, che rappresentano entità differenti nel modello (rispettivamente *Category* e *User*). Ebean **supporta pienamente la transitività**.

5.2.2 Elementi di complessità

Ebean, sebbene più semplice delle alternative più pesanti, necessita di una fase di configurazione iniziale. In primo luogo, utilizzando **Maven**, è necessario importare tutte le dipendenze.

La connessione con il database segue delle regole definite nel file *application.yaml*.

```
1  datasource:
2    db:
3      username: ${DB_USER}
4      password: ${DB_PASSWORD}
5      url: jdbc:postgresql://${DB_HOST}:${DB_PORT}/${
        DB_DATABASE}
6
7  ebean:
8    docstore:
9      url: http://${ELASTIC_HOST}:${ELASTIC_PORT}
10     active: true
11     generateMapping: true
12     dropCreate: false
13     migration:
```

```
14      run: true
```

In questo caso *datasource* definisce i parametri di connessione per il database primario, un DBMS PostgreSQL, recuperando *username*, *password*, *host*, *porta* e *nome* del database dalle variabili d'ambiente.

Il campo *docstore*, sotto *ebean*, definisce invece la connessione con l'indice *Elasticsearch* per la ricerca full-text.

L'integrazione con il logger non è in questo caso automatica, come per Spark, ma è necessario aggiungere alcune righe di codice nel file di configurazione del logger, nel nostro caso *logback.xml*.

Un'ulteriore elemento di complessità è la stessa documentazione, che non sempre si riferisce all'ultima versione disponibile del framework e soprattutto non copre con adeguata profondità alcune funzionalità, rendendo più complicata la configurazione.

Dato che il progetto non può contare su una community particolarmente numerosa¹ non sempre è facile recuperare informazioni online.

È importante tener conto delle **limitazioni note** in fase di valutazione dei requisiti funzionali: Ebean rappresenta sicuramente una soluzione leggera e adeguatamente versatile, ma non sempre è in grado di soddisfare tutte le esigenze dello sviluppatore.

5.3 Elasticsearch

Elasticsearch è un motore di ricerca estremamente potente, specializzato nella ricerca *full-text* ed integrabile con Ebean.

È implementato come un database *document oriented*, che indicizza i docu-

¹al momento della stesura di questo documento (settembre 2021), il progetto su Github conta 1200 star e 230 fork.

menti forniti in modo che la ricerca e il recupero risultino molto veloci.

L'integrazione di Elasticsearch con Ebean rappresenta un grande vantaggio nel caso di studio considerato: è possibile utilizzare **molte** delle funzionalità offerte attraverso l'uso di query ORM, di fatto evitando l'implementazione manuale di una ricerca full-text sul database (complicata nella forma e peggiore nelle prestazioni).

```
1 /**
2  * Returns the list of museums found with the full-text search
   given a query.
3  *
4  * @param query      a string query for the full-text search
5  * @return a List of Museum objects
6  */
7 public List<Museum> fullText(String query) {
8     MultiMatch multiMatch =
9         MultiMatch.fields("description", "name").opAnd().type(
10            MultiMatch.Type.CROSS_FIELDS);
11     PagedList<Museum> museumList = DB.find(Museum.class)
12         .text()
13         .multiMatch(query, multiMatch)
14         .select("museumId, name")
15         .setMaxRows(100)
16         .findPagedList();
17     return museumList.getList();
18 }
```

Capitolo 6

Prospettiva di progettazione

L'applicazione dei principi generali del modello di transizione, insieme all'utilizzo dei framework citati, nel passaggio da applicazione desktop ad applicazione web, permette di semplificare in maniera significativa la struttura dei package.

Il *class diagram* in figura, relativo all'applicazione di partenza è stato modificato aggiungendo dei riquadri colorati: le classi in verde sono mantenute durante la transizione, mentre quelle in rosso sono state eliminate.

È evidente come l'utilizzo di framework semplifichi notevolmente il diagramma, in primo luogo integrando funzionalità che precedentemente dovevano essere implementate dallo sviluppatore e, come vedremo nel capitolo successivo, riducendo sensibilmente la quantità di codice richiesta.

Analizzando come nostro solito a partire dal livello più alto, notiamo che il **service layer** non è incluso nel diagramma delle classi: come anticipato infatti nel capitolo sulla metodologia, in questo livello non abbiamo una *ristrutturazione*, ma una vera e propria *trasformazione*, dovuta al cambia-

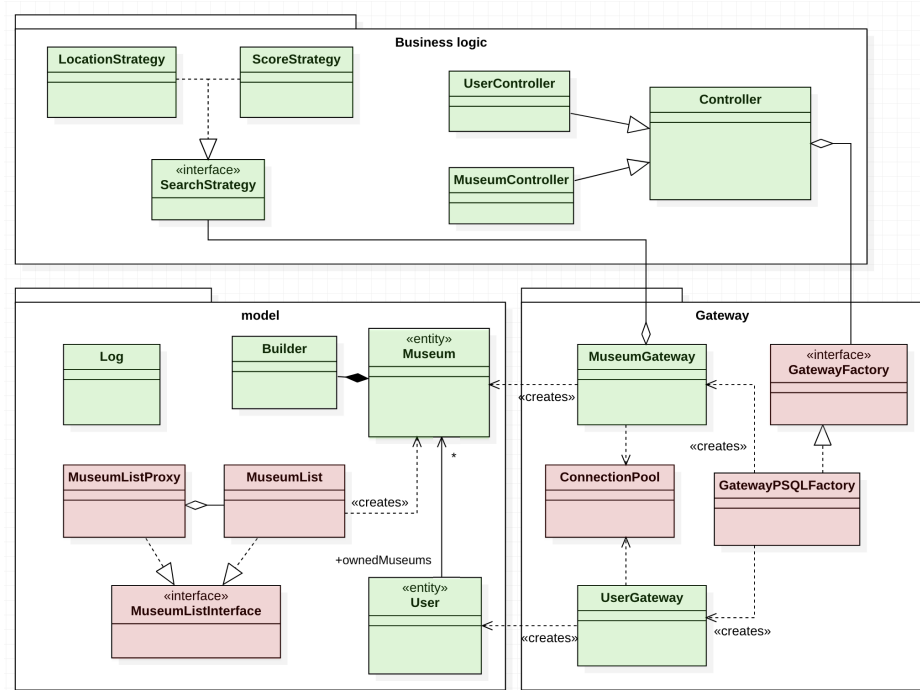


Figura 6.1: Class Diagram

mento di *contesto* (da desktop a web) e di *paradigma* (sono esposti servizi e non viste).

Le *view* di *Swing* sono sostituite da dagli endpoint gestiti da *Spark*, l'interfaccia grafica è completamente eliminata.

La **business-logic**, invece, rimane identica nella struttura, con differenze solo a livello di codice.

I cambiamenti sono evidenti nel **data layer**, dove l'introduzione di *Ebean* permette di ridurre significativamente il numero di classi necessarie:

- nel package **gateway** viene meno la necessità di un *ConnectionPool*, le connessioni sono incapsulate in maniera trasparente da Ebean.

Anche il *GatewayFactory* (e la sua realizzazione *GatewayPSQLFactory*), che fornisce metodi per istanziare i vari tipi di gateway (in questo caso un gateway in grado di relazionarsi con un database PostgreSQL).

SQL), non è più necessario: Ebean ha un'elevata portabilità rispetto alla tecnologia DBMS utilizzata e permette, attraverso l'utilizzo di query ORM, di nascondere il linguaggio utilizzato per interrogare la base dati.

- il **modello** stesso viene semplificato. Per memorizzare e operare con la lista di musei che costituisce il risultato di una ricerca era stata creata una classe apposita **MuseumList**. Dato che questa lista poteva essere molto lunga e veniva creata contemporaneamente all'oggetto che la contiene, era stato deciso di utilizzare il pattern **Proxy** per implementare la **lazy initialization**. Tramite il proxy, infatti, la lista dei risultati è creata solo quando è realmente richiesta, ovvero nel momento in cui viene effettuata una ricerca. Oltre a gestire il *lifecycle* degli oggetti di tipo **MuseumList**, questo proxy permette anche di fare **caching**, restituendo la stessa istanza della lista nel caso l'utente richiedesse la visualizzazione di più risultati.

Sebbene il caching e la lazy initialization rappresentino ancora dei requisiti necessari nella web application, il *MuseumListProxy* e le classi ad esso collegate non sono più necessarie poiché tutte le feature sono implementate nativamente da Ebean.

Ci saremmo potuti aspettare che il modello non presentasse tutte queste modifiche, ma che piuttosto mantenesse la stessa struttura nel passaggio da web a desktop application. È da specificare però che il cambiamento è dovuto piuttosto all'introduzione di un framework ORM: non è più necessario che lo sviluppatore metta in vita delle collezioni specializzate.

D'altro canto, inserire questi *comportamenti* nel modello piuttosto che nella business-logic è anch'essa una scelta architetturale (valida se as-

sociata ad una prospettiva di sviluppo in Java Standard Edition).

Capitolo 7

Prospettiva di implementazione

Oltre che nella struttura, anche da un punto di vista implementativo possiamo osservare una semplificazione: il codice sorgente risulta **meno corposo** in primo luogo a causa dell'eliminazione dell'interfaccia grafica, ma anche grazie alla gestione del DBMS da parte di Ebean.

Soprattutto nel data layer notiamo una maggiore **facilità di interpretazione**: le query SQL sono adesso espresse con metodi Java ed il modello relazionale del database trova una corrispondenza nel domain model.

7.1 Implementazione del Service Layer

7.1.1 Web Application

Nel service layer della web application non rimane alcuna traccia del software di partenza: i servizi sono esposti sfruttando i metodi della libreria di Spark.

```
1 path("/museum", () -> {  
2   get("/:museumId", (request, response) -> {
```



```
3     response.type("application/json");
4     return museumController.getMuseumResponse(request.params(
5         ":museumId"), response);
6 });
7
8 put("/:museumId", (request, response) -> {
9     response.type("application/json");
10    return museumController.saveMuseumResponse(request.body()
11        , response, request.params(":museumId"), request.headers("
12        Authorization"));
13 });
14
15 post("", (request, response) -> {
16     response.type("application/json");
17     return museumController.saveMuseumResponse(request.body()
18         , response, null, request.headers("Authorization"));
19 });
20 });
```

In particolare, i vari metodi:

- **intercettano la richiesta.** In base al metodo HTTP utilizzato dal client ed al path, la richiesta viene gestita da uno dei metodi del service layer attivando la sua funzione di callback. Se non fosse definita alcuna route compatibile con la richiesta dell'utente, verrebbe restituita una risposta di errore con codice di stato *404*;
- **demandano al controller** la costruzione del body. All'interno della funzione di callback non è presente logica di programma, ogni operazione deve essere eseguita nel business-logic layer;
- **restituiscono la risposta.** Il body della risposta è in formato JSON (`response.type("application/json")`).

7.2 Implementazione del Controller

7.2.1 Desktop Application

```
1 public Museum getMuseum(int museumId) {
2     Museum museum = null;
3     try {
4         museum = gatewayFactory.getMuseumGateway().getMuseum(
5             museumId);
6     } catch (SQLException e) {
7         handleError(e);
8     } catch (IllegalArgumentException e) {
9         logger.warning(Log.getStringStackTrace(e));
10        JOptionPane.showMessageDialog(null, e.getMessage());
11    }
12    return museum;
13 }
```

Nella desktop application il controller

- **interagisce con il gateway.** Le entità sono recuperate dal database attraverso la mediazione del data layer;
- **gestisce le eccezioni**, che possono essere lanciate sia dai metodi dello stesso package che da metodi del livello inferiore.

La corretta gestione prevede che l'errore sia riportato nel *file dei log* e che un messaggio venga presentato all'utente sotto forma di pop-up.

7.2.2 Web Application

```
1 public Museum getMuseum(String museumId) throws
2     ResponseException {
3     long parsedId;
```

```
3  try {
4      parsedId = Long.parseLong(museumId);
5  } catch (NumberFormatException e) {
6      logger.error(Log.getStringStackTrace(e));
7      throw new ResponseException("Provided museum id is not a
      number!", 422);
8  }
9  Museum museum;
10 try {
11     museum = museumGateway.getMuseum(parsedId);
12 } catch (Exception e) {
13     logger.error(Log.getStringStackTrace(e));
14     throw new ResponseException("Something went wrong!", 500)
15     ;
16 }
17 if (museum == null) {
18     logger.warn("Museum with id " + museumId + " not found!")
19     ;
20     throw new ResponseException("Museum not found.", 404);
21 }
22 return museum;
23 }
```

La logica del controller **non cambia** nella web application: le responsabilità continuano ad essere quella di interagire con il livello inferiore per il recupero delle entità e la gestione delle eccezioni.

Si rendono tuttavia necessari nuovi metodi per la costruzione della risposta HTTP, sia per la scelta dei codici di stato che per la generazione del body in formato JSON.

```
1 /**
2  * Returns a JSON encoded museum
```

```
3 * based on the provided museum id.
4 *
5 * @param museumId a string which should contain a museum id
6 * @param response the HTTPS Spark response
7 * @return the Spark response with a Museum object as the JSON
    body
8 */
9 public Response getMuseumResponse(String museumId, Response
    response) {
10     Message message = new Message();
11     Museum museum;
12     try {
13         museum = getMuseum(museumId);
14     } catch (ResponseException e) {
15         return generateErrorResponse(response, e.getMessage(), e.
            getStatusCode());
16     }
17     response.body(gson.toJson(encapsulateJsonObject(gson.
        toJsonTree(message), "museum", DB.json().toJson(museum))))
        ;
18     return response;
19 }
```

In caso di errore, la risposta HTTP ha un codice di stato diverso da *200 OK* e presenta nel body un messaggio per l'utente. In caso contrario l'entità richiesta viene restituita, sfruttando i *metodi di serializzazione automatica di Ebean*.

7.3 Implementazione del Gateway

Nel data layer sono invece molto evidenti le differenze a livello di codice. Consideriamo a titolo di esempio il metodo per il recupero del museo nel database, dato il suo id.

7.3.1 Desktop Application

```
1 public Museum getMuseum(int museumId) throws SQLException {
2     Connection connection = ConnectionPool.getConnection();
3     logger.info("Retrieving museum with id " + museumId + "...")
4         );
5     ResultSet resultSet = null;
6     Museum museum;
7
8     try (PreparedStatement statement = connection.
9         prepareStatement("SELECT museum_id, m.name, wiki_link, " +
10         "website, location, lat, lng, description, address,
11         string_agg(c.name, ', '), thumb_url " +
12         "FROM museums m LEFT JOIN museum_category mc on m.museum_id
13         = mc.fk_museum_id " +
14         "LEFT JOIN categories c on c.category_id = mc.
15         fk_category_id " +
16         "LEFT JOIN museum_image mi on m.museum_id = mi.fk_museum_id
17         WHERE museum_id = ? " +
18         "GROUP BY museum_id, m.name, wiki_link, website, location,
19         lat, lng, description, address, thumb_url")) {
20         statement.setInt(1, museumId);
21         logger.fine("QUERY: " + statement.toString());
22         resultSet = statement.executeQuery();
23         if (resultSet.next()) {
```

```
17     museum = new Museum.Builder(resultSet.getInt(1),
18     resultSet.getString(2))
19     .wikiLink(resultSet.getString(3))
20     .website(resultSet.getString(4))
21     .location(resultSet.getString(5))
22     .lat(resultSet.getDouble(6))
23     .lng(resultSet.getDouble(7))
24     .description(resultSet.getString(8))
25     .address(resultSet.getString(9))
26     .categories(resultSet.getString(10))
27     .imageUrl(resultSet.getString(11))
28     .build();
29 } else throw new IllegalArgumentException("Provided id
30 doesn't match any museum in the database");
31 } finally {
32     try {
33         if (resultSet != null) resultSet.close();
34     } catch (Exception ignored) {
35     }
36     ConnectionPool.releaseConnection(connection);
37 }
38 logger.info("Museum retrieved");
39 return museum;
40 }
```

Il gateway si occupa nella desktop application di:

- **gestire le connessioni**, gli **statement** ed i **result set**. Le connessioni devono essere acquisite da un Connection Pool (`Connection connection = ConnectionPool.getConnection()`) ed infine rilasciate (`ConnectionPool.releaseConnection(connection)`), gli statement ed i result set esplicitamente chiusi (`resultSet.close();`);

- **interrogare il database** tramite una *complicata* query SQL;
- **costruire l'entità Museo**, in maniera non banale, a partire dalle informazioni recuperate dalla base dati.

7.3.2 Web Application

```
1 public Museum getMuseum(long museumId) {  
2     return new QMuseum()  
3         .museumId.eq(museumId)  
4         .categories.fetch(QCategory.alias().name)  
5         .owners.fetch(QUser.alias().username)  
6         .findOne();  
7 }
```

Lo stesso metodo è *enormemente* semplificato grazie ad Ebean:

- le connessioni, gli statement ed i result set sono gestiti internamente;
- la query ORM è rappresentata da una *concatenazione di metodi Java*;
- il *mapping* del risultato ad un'entità Museo viene fatto automaticamente.

Capitolo 8

Conclusioni

Chiudiamo questa trattazione con una serie di riflessioni relative ad alcune scelte (architetture) fatte durante l'implementazione del caso di studio e nel processo di transizione. In particolar modo poniamo l'attenzione sulla possibilità di *impiegare i DTO* per il trasferimento delle informazioni e sulle *implicazioni dal punto di vista delle prestazioni* dell'utilizzo di un framework ORM per l'astrazione della base dati.

8.1 Utilizzo dei Data Transfer Object

Il *Data Transfer Object* è un pattern architeturale che aggrega dati per il trasferimento. Si comporta come una struttura dati, ma contiene anche i meccanismi di serializzazione e deserializzazione. Dato che il suo scopo è quello di trasferire solo i dati necessari ad un chiamante remoto, cercando di limitare anche il numero di richieste, il DTO deve poter contenere informazioni (anche parziali) di una o più risorse.

Nel caso pratico dell'applicazione di ricerca e gestione dei musei l'utilizzo dei DTO può essere valutato, ma non risulta necessario: ad ogni richiesta il client

si aspetta di ricevere solamente informazioni relative ad un museo (o ad una lista di musei) senza dover integrare informazioni da altre classi di oggetti, se non un messaggio facoltativo che può accompagnare la risposta. A questo scopo Ebean già fornisce un comodo metodo di serializzazione JSON delle entità recuperate dal database (basato su *Jackson*) che include solamente gli attributi effettivamente selezionati dalla query ORM.

In ogni caso, per flessibilità, la classe *Controller*, che si occupa di generare il body della risposta HTTP, presenta un semplice metodo per ampliare l'albero JSON con elementi esterni, come ad esempio un messaggio personalizzato da inviare al front-end:

```
1  private JsonElement encapsulateJsonObject(JsonElement
    jsonElement, String name, String json) {
2      JsonObject jsonObject = gson.fromJson(json, JsonObject.
        class);
3      jsonElement.getAsJsonObject().add(name, gson.toJsonTree(
        jsonObject));
4      return jsonElement;
5  }
6
7  encapsulateJsonObject(gson.toJsonTree(message), "museum",
    DB.json().toJson(museum))
```

Nel caso il client avesse richiesto più informazioni, come ad esempio dati relativi al luogo in cui si trova un museo (eventualmente sempre recuperabili da un database), il metodo *encapsulateJsonObject* avrebbe permesso comunque di combinare le informazioni, sfruttando sempre la serializzazione di Ebean: anche in un contesto più generale, per semplicità, il pattern DTO può essere sostituito.

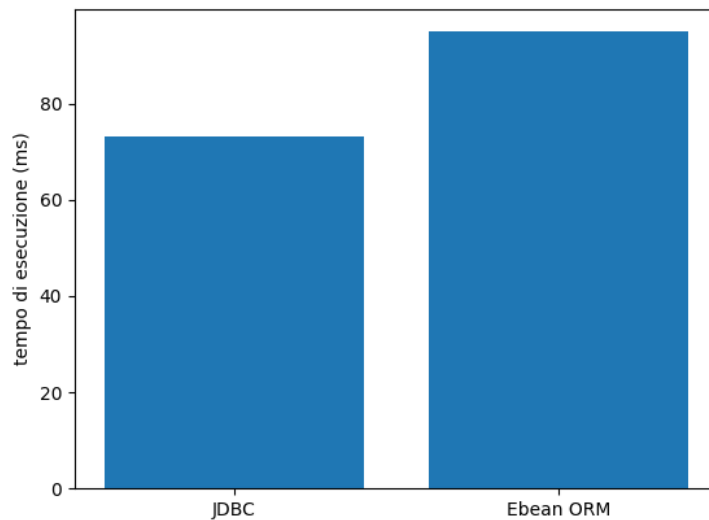


Figura 8.1: Confronto del tempo di esecuzione nel recupero di un'entità museo dal database

8.2 Analisi delle prestazioni

Un framework ORM, indipendentemente dai benefici che offre, rappresenta di fatto un *ulteriore livello* che divide il data layer dal DBMS. In generale questo aumento di complessità dell'architettura si traduce durante l'esecuzione in un maggior numero di chiamate a funzione, dovute ad esempio alla traduzione delle query ORM in query SQL, oppure al mapping delle entità sul database.

L'analisi delle prestazioni permette di quantificare il calo della performance: analizzando l'azione che più frequentemente viene eseguita, ovvero il recupero di un'entità museo dal database possiamo però osservare che l'utilizzo di Ebean ORM piuttosto che del semplice connettore JDBC si traduce in un ritardo nella risposta di circa *20 ms* (un tempo decisamente accettabile nel contesto di un'applicazione web).

È altresì possibile evidenziare situazioni in cui l'integrazione del framework permetta piuttosto di migliorare (anche in maniera significativa) le prestazioni del software.

Consideriamo in particolare l'integrazione con Elasticsearch per la ricerca full-text. I database relazionali più famosi (come PostgreSQL) supportano la creazione di *indici full-text*, ma questo tipo di ricerca non rappresenta il loro fine principale: il risultato è una query di interrogazione particolarmente complicata¹ e un tempo di esecuzione che cresce linearmente all'aumentare delle parole ricercate. Invece, sfruttare un database specificatamente studiato per la ricerca full-text permette di avere prestazioni decisamente migliori. In questo caso il tempo di esecuzione non cresce linearmente, ma rimane costante; in realtà il trend sembra essere inverso: la ricerca con poche parole chiave impiega più tempo di quella con molti termini. Effettivamente query di ricerca molto specifiche trovano poche corrispondenze nel database mentre con pochi termini ho la possibilità di ottenere molti risultati di ricerca: nell'esecuzione teniamo conto anche del tempo necessario ad istanziare tutte le entità (sebbene siano restituiti solamente i campi principali, ovvero *nome* del museo ed *id*).

¹il caso di studio necessita di dare un peso ad ogni risultato di ricerca ponderato anche in base alla frequenza di ogni parola chiave in **tutti** i documenti. Ad esempio il termine "*museo*" deve avere un peso inferiore rispetto ad un termine più specifico come "*arte orientale*". Tradurre questo requisito a livello di query SQL non è banale e aumenta decisamente la complessità dell'interrogazione

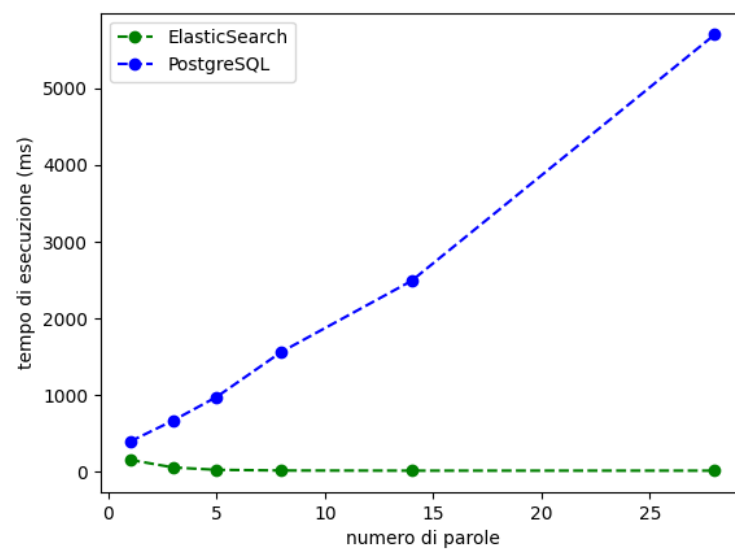


Figura 8.2: Confronto del tempo di esecuzione nella ricerca full-text

Bibliografia

- [1] Web application architecture: How the web works. <https://www.altexsoft.com/blog/engineering/web-application-architecture-how-the-web-works/>, 2019.
- [2] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [3] Fabio Luccioletti. Museum finder - rest. <https://github.com/Satap27/MuseumFinder-REST>, 2021.
- [4] Fabio Luccioletti and Gianmarco Pastore. Museum finder. https://github.com/IngInf/Museum_Finder, 2021.