DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria dei Sistemi Informativi

Refactoring di un Software per la Prenotazione di Servizi Sanitari

Refactoring of a Software for Healthcare Services' Booking System

UNIVERSITÀ
DI PARMA

Relatori
Prof. Amoretti Michele

Correlatori:
Prof. Prati Andrea
Dott. Ing. Strozzi Fabio

Laureando
Daniele Pellegrini

Anno Accademico 2019 - 2020

Alla mia famiglia, da sempre centro gravitazionale della mia vita.

Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn.

Stephen Hawking

Everybody in this country should learn to program a computer, because it teaches you how to think

Steve Jobs

Ringraziamenti

Sommario

Il presente elaborato ha come obiettivo la **reingegnerizzazione di un software utilizzato per la prenotazione di servizi sanitari**. A seconda della struttura ospedaliera presso cui si prenota, i servizi offerti da questo applicativo sono molteplici e di diverso tipo: esami di laboratorio, esami del sangue, visite mediche o accettazioni. Il primo scopo di questa applicazione è dunque quello di velocizzare l'accesso ai servizi, controllandone l'affluenza. Oltre alla riduzione del contatto interpersonale, l'ottimizzazione di software come questo può ridurre il rischio di assembramento e una migliore profilazione dell'utente con cui il personale e gli altri clienti entrano in contatto.
Il COVID19 ha portato questa esigenza, che inizialmente era una comodità, ad essere a tutti gli effetti **una necessità** e un requisito fondamentale per le aziende che richiedono questo tipo di servizi: per la loro gestione interna e per garantire la sicurezza dei loro clienti, a maggior ragione per quelle nel settore sanitario. Il sistema è attualmente online e operativo, ma non ottimizzato per il suo funzionamento.
Con un'azione di reverse engineering si andranno ad analizzare i punti critici del sistema, a studiarne il traffico e l'effettivo comportamento. Non si agirà direttamente sul frontend, la parte visibile dall'utente finale e con la quale interagisce, ma *sul backend*, la parte che elabora i dati e che ha il compito di interfacciarsi con il database, dove vengono salvati i dati. Il fine ultimo di quest'opera di reingegnerizzazione sarà la costruzione di un **nuovo layer di Rest APIs** dietro un reverse proxy. Il sistema attuale non presenta un layer di API, pertanto le chiamate dal frontend al backend vengono trasmesse in modo contorto e poco efficiente: la banalità del sistema monolitico potrebbe essere causa di un attacco informatico o malfunzionamenti, per questo si è optato per un'architettura di microservizi. Questo nuovo strato verrà inizialmente affiancato al sistema esistente, fino a quando un successivo sviluppo ne renderà possibile una completa migrazione. La parte di backend, attualmente scritta in PHP, verrà riscritta in Java, utilizzando la JDK 15 e il framework Spring, mentre per una gestione più ottimale del progetto si utilizzerà Maven. Con la reingegnerizzazione il sistema potrà ottenere nuove funzionalità richieste dalle strutture affiliate e migliorare quelle già implementate, l'importante è che **rimanga coerente con il suo attuale funzionamento**: il nuovo strato che si inserirà tra frontend e backend dovrà garantire il corretto funzionamento di tutte le operazioni già presenti.
In una società sempre più frenetica e in continuo movimento, l'idea di Zerocoda ha preso piede velocemente. Sulla sua base, si è pertanto deciso di creare un'applicazione analoga per i servizi commerciali, un'esigenza diversa, ma vicina alla precedente. L'idea è già stata sviluppata partendo proprio da questo software, pertanto il layer di API che si andrà a realizzare costituisce il primo passo sulla strada che porterà alla fusione delle due.

Indice

Elenco delle figure

Capitolo 1

Stato dell'arte

In questo capitolo si illustrano le principali tecnologie adottate, spiegandone il funzionamento e le motivazioni che hanno portato alla loro scelte, preferendole alle alternative.

1.1 Programmazione Object-Oriented

La Programmazione ad Oggetti rappresenta, senza dubbio, il modello di programmazione più diffuso ed utilizzato degli ultimi dieci anni. Le vecchie metodologie come la programmazione strutturata e procedurale, in auge negli anni settanta e punto di riferimento per lo sviluppo software, sono state lentamente ma inesorabilmente superate a causa degli innumerevoli vantaggi che sono derivati dall'utilizzo del nuovo paradigma di sviluppo. Via via che gli orizzonti della programmazione diventavano sempre più ampi, si andavano evidenziando i limiti delle vecchie metodologie. In particolare, un programma procedurale mal si prestava a realizzare il concetto di *componente software*, ovvero di un prodotto in grado di garantire le caratteristiche di *riusabilità*, *modificabilità* e *manutenibilità*.

1.1.1 Le origini

Per comprendere meglio il significato di *programmare per oggetti* è necessario comprendere cosa significhi non farlo. In questa sezione si propone con una breve analisi dei precursori dell'*Object Oriented Programming*.

Programmazione non strutturata [1.1]

In questo paradigma il programma è costituito da *un unico blocco di codice detto "main"* dentro il quale vengono manipolati i dati in maniera totalmente sequenziale. Qui i dati sono rappresentati soltanto da variabili di tipo globale, ovvero visibili da ogni parte del programma ed allocate in memoria per tutto il tempo che il programma stesso rimane in esecuzione. È facile intuire come ciò sia un notevole svantaggio. Questo tipo di programmazione comporta ridondanza nel codice e un'enorme spreco di risorse del sistema.

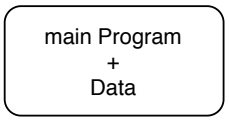

Figura 1.1. Programmazione non strutturata

Programmazione procedurale[1.2]

Il concetto base di questo tipo di programmazione è quello di raggruppare i pezzi di programma ripetuti in porzioni di codice utilizzabili e richiamabili ogni volta che se ne presenti l'esigenza. Le porzioni di codice con queste caratteristiche prendono il nome di *procedure*. Ogni procedura può essere vista come un *sottoprogramma che svolge una ben determinata funzione* e che è visibile e richiamabile dal resto del codice. La programmazione procedurale rappresenta un notevole passo in avanti rispetto a quella non strutturata, in quanto ne supera i limiti di ridondanza e garantisce una migliore gestione della memoria di sistema. Il vantaggio di una procedure sta nell'utilizzo dei parametri, allocati in memoria solo nel momento in cui una quest'ultima viene chiamata. Il *main continua ad esistere* ma al suo interno presenta esclusivamente le invocazioni alle procedure definite dal programma.

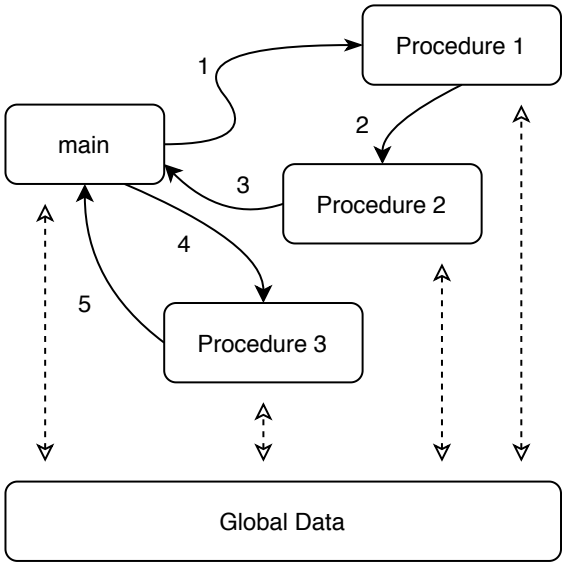

Figura 1.2. Programmazione procedurale

Quando una procedura ha terminato il suo compito, il controllo ritorna nuovamente al main (o alla procedura che ne ha effettuato l'invocazione) che esegue una nuova chiamata ad un'altra procedura fino alla terminazione del programma.

Programmazione modulare[1.3]

Questo paradigma rapprenta un ulteriore passo avanti rispetto ai precedenti. La programmazione modulare risponde all'esigenza di poter *riutilizzare le procedure messe a disposizione da un programma in modo che anche altri programmi ne possano trarre vantaggio.* L'idea alla base di questo paradigma è quella di raggruppare le procedure aventi un dominio comune (ad esempio, procedure che eseguono operazioni matematiche) in moduli separati. Quando si parla di **librerie di programmi**, in sostanza si fa riferimento proprio a moduli di codice indipendenti che ben si prestano ad essere riutilizzati in svariati programmi.

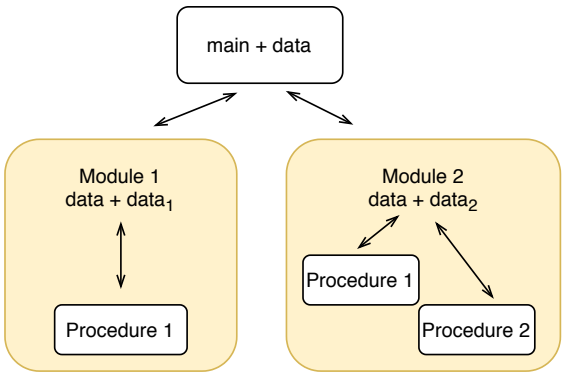

Figura 1.3. Programmazione modulare

Con questo paradigma un singolo programma non è più costituito da un solo file (in cui è presente il main e tutte le procedure) ma da diversi moduli. Un singolo modulo può contenere anche dei dati propri che, in congiunzione ai dati del main, vengono utilizzati all'interno delle procedure in essi contenuti.

1.1.2 Definizione di Object Oriented Programming

Cosa si intende quindi quando si parla di programmazione a oggetti? Dalla rivista MCmicromputer, una delle riviste storiche trattanti argomenti di informatica in Italia, sappiamo che con il termine programmazione orientata agli oggetti (da qui in seguito *OOP*, in riferimento all'acronimo inglese) si pensa a un insieme di dati come a un singolo oggetto. [1] Più oggetti possono interagire vicendevolmente, scambiandosi messaggi ma mantenendo ciascuno il proprio stato e i propri dati. Questa *rivoluzione del metodo di programmazione* cambia l'approccio mentale all'analisi progettuale, ma non rinuncia ai vantaggi fino ad ora introdotti dai paradigmi precedenti, in particolare a quelli derivanti dall'utilizzo dei moduli. L'idea alla base di questo principio risiede, in buona parte, nel mondo reale. Un *oggetto* è tipicamente un oggetto del mondo reale. In questo paradigma non ha importanza l'implementazione del codice ma, piuttosto, **le caratteristiche e le azioni** che un componente software è in grado di svolgere e che mette a disposizione di altri oggetti.

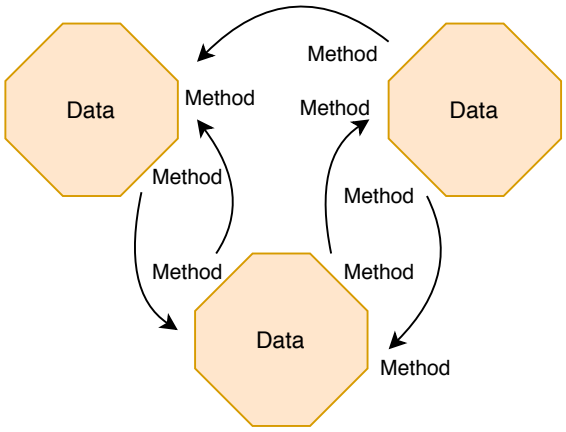

Figura 1.4. Colloquio tra oggetti

Nella figura [1.4] gli oggetti sono rappresentati dagli esagoni, che contengono i dati e comunicano attraverso metodi. Nella OOP, i dati (o caratteristiche dell'oggetto), prendono il nome di **proprietà**, mentre le azioni che essi possono fare **metodi**. Le proprietà vengono utilizzate dai metodi per eseguire determinate operazioni. Con l'espressione **pensare "ad oggetti"** quindi si identificano gli oggetti che entrano in gioco nel programma che si vuole sviluppare, gestendone l'interazione degli uni con gli altri.

1.1.3 Java | Oracle

> «The fact that you know Java doesn't mean that you have the ability to transform that knowledge into well-designed object oriented systems»
>
> Paul R. Reed

.

La nascita del linguaggio Java alla fine del XX secolo segnò un punto di svolta per la programmazione a oggetti. La prima grande rivoluzione introdotta da questo linguaggio fu quella di liberare il programmatore dall'onere della gestione della memoria, che prima era gestita mediante l'utilizzo dei puntatori. Grazie a un sistema chiamato **garbage collector**, Java è in grado di assegnare e rilasciare automaticamente la memoria in base alla gestione del programma. La seconda rivoluzione introdotta è legata alla Java Virtual Machine (JVM), grazie alla quale i programmi non sono più compilati in codice macchina, ma in una sorta di linguaggio macchina "intermedio" (chiamato bytecode) che non è destinato ad essere eseguito direttamente dall'hardware ma che deve essere, a sua volta, interpretato da un secondo programma, la macchina virtuale appunto. In questo modo lo stesso codice può essere eseguito su più piattaforme semplicemente trasferendo il bytecode (non il sorgente) purché sia disponibile una JVM. Questo concetto prende il nome di **WORA**, *write once, run everywhere*. [2] Java fu incorporato da diversi web browser per permettere l'utilizzo delle *applet*, un'applicazione che può essere avviata dall'utente eseguendo il codice scaricato da un server web remoto.

Perché Java?

Il linguaggio di programmazione è stata la prima decisione per un'ottimale azione di refactoring dell'applicazione e Java si è dimostrato essere il migliore per l'obiettivo del progetto. Il rifacimento del backend è il primo passo sulla strada che porterà Zerocoda a un'evoluzione con conseguente inserimento in un panorama più ampio. Per questo motivo, la quasi totale indipendenza di Java dalla piattaforma hardware di esecuzione è una necessità chiave per l'applicazione, che così acquisisce una maggiore scalabilità.

La popolarità

Prendendo in analisi i grafici offerti da *TIOBE Programming Community Index* [3] contenente in ordine i linguaggi di programmazione più utilizzati, Java sembrerebbe essere al primo posto. Si tratta di un linguaggio facilmente manutenibile e dotato di molta documentazione che ne facilita l'apprendimento. Con le diverse librerie e framework compatibili è diventato il linguaggio di programmazione a oggetti più utilizzato al mondo, primo anche al precursore C++, dove la gestione della memoria e la curva di apprendimento possono rappresentare un problema. Java è stato adottato per la realizzazione del backend di diversi siti web di rilievo, come il sito d'asta e vendita online *eBay*.

Le motivazioni

Uno dei più grandi vantaggi offerti da Java è proprio la sua capacità di addattamento ad aggiornamenti e modifiche del software in corso d'opera. Per capire meglio il perchè di questa scelta, facciamo riferimento all'esempio in figura [1.5]:

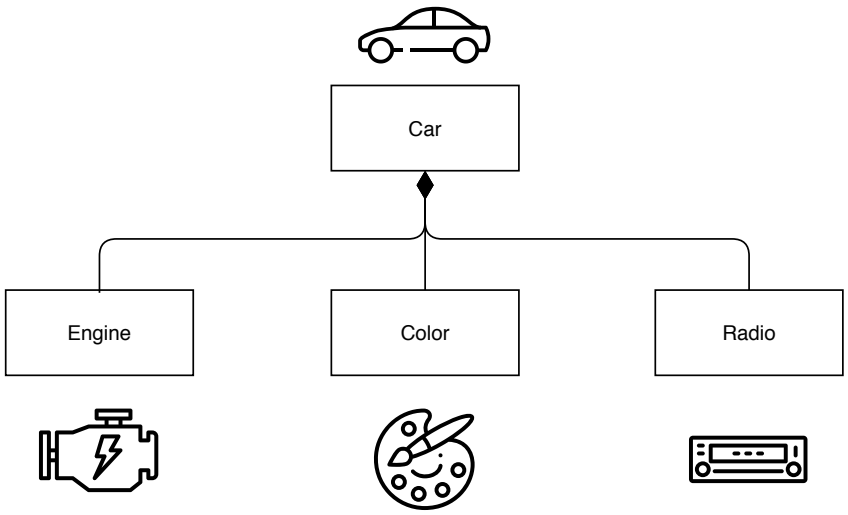

Figura 1.5. Esempio di un Sistema Orientato agli Oggetti

Si pensi ad un rivenditore di auto che ha vari veicoli nel suo parco mezzi. Ogni veicolo è un `oggetto`, ma ognuno ha caratteristiche diverse denominate `classi`, che nel nostro esempio sono i diversi modelli, motori, colori della carrozzeria. Un cliente sceglie una macchina rossa, ma desidera aggiungere un impianto stereo migliore. La nuova macchina erediterà tutte le caratteristiche dell'oggetto "car" lasciando al programmatore il compito semplificato di modificare solamente la classe "radio" piuttosto che costruire da capo l'intero veicolo. Questo è ciò che rende Java la piattaforma ideale per i telefoni cellulari, i siti web, le console di gioco e qualsiasi applicazione che richieda aggiornamenti e modifiche frequenti.

1.2 Database Relazionali

Nello svolgimeto di ogni attività, sia a livello individuale sia in organizzazioni di ogni dimmensione, sono essenziali la disponibilità di informazioni e la capacità di gestirle in modo efficiente [4]. Un database relazionale è un tipo di database di archiviazione che fornisce accesso a data points tra i quali sussistono relazioni predefinite e che soddisfa queste esigenze. I database relazionali sono basati sul modello relazionale, un modello di rappresentazione dati semplice e diretto basato sull'utilizzo di tabelle.

1.2.1 Modello Relazionale

Il modello relazionale si basa su due concetti: **relazione** e **tabella**. La nozione di *relazione* proviene dalla matematica, in particolare dalla teoria degli insiemi, mentre il concetto di *tabella* è intuitivo. Il punto di forza del modello di database relazionale è l'uso delle tabelle, che permettono di archiviare informazioni strutturate e accedervi, risultando comprensibili anche per gli utenti finali. [1.6]

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	25/11/1996
485745	Neri	Fabio	23/04/1997

Studente	Corso	Voto
276545	Analisi	28
485745	Chimica	27

Figura 1.6. Esempio di un Database Relazionale

- l'ordine delle colonne e delle righe all'interno di una tabella è insignificante
- non possono esistere due righe identiche, ogni riga può essere contrassegnata con un identificatore univoco (chiave principale) che ne faciliti la distinzione dalle altre
- ogni colonna in una tabella ha un nome univoco e contiene un solo tipo di dato

- un campo (o cella) può contenere un solo valore effettivo di un attributo
- le righe di tabelle diverse possono essere correlate utilizzando chiavi esterne

1.2.2 Struttura

Secondo il modello relazionale, le strutture dei dati logici, ovvero tabelle di dati, viste e indici, sono separate dalle strutture di storage fisiche. Grazie a questa separazione, gli amministratori di database possono gestire lo storage fisico dei dati senza compromettere l'accesso a tali dati come struttura logica. Questo permette di accedere ai dati in modi diversi senza riorganizzare le tabelle e di ottenere quindi **l'indipendenza fisica dei dati**.

1.2.3 Database non Relazionali

Questo tipo di struttura dati si differenzia da quella appena presentata dal momento che non richiede uno schema fisso. La base su cui poggia tutta la costruzione dei database di questo tipo non è costituita da tabelle di dati ma da documenti. La forza di questo principio è proprio che tutto quello che serve all'applicazione risiede nel documento già precompilato. Si evita le frammentazione dell'informazione e la sua ricostruzione, con i rischi di perdere dati o averne di corrotti. Un aumento di flessibilità che velocizza le operazioni e offre risposte più veloci all'utente finale.

```
{
    "276545" : {
        "cognome" : "Rossi",
        "nome" : "Maria",
        "data_di_nascita" : "25/11/1996",
        "esami" : [
            {
                "corso" : "Analisi",
                "voto" : 28,
            }
        ]
    },
    "485745" : {
        "cognome" : "Neri",
        "nome" : "Fabio",
        "data_di_nascita" : "23/04/1997",
        "esami" : [
            {
                "corso" : "Chimica",
                "voto" : 27,
            }
        ]
    },
}
```

Figura 1.7. Esempio di documento di un database non relazionale

1.2.4 Le differenze

Un database non relazionale (chiamato anche *NoSQL*) è preferibile quando si ha a che fare con una grande quantità di dati. Questa sua struttura, molto aperta ad aggiunte e scalabile orizzontalmente, rappresenta una grande fattore di rischio per un problema che nei database relazionali è gestito in maniera ottimale: la duplicazione dei dati. Un aspetto che tuttavia depone a favore dei database non relazionali lo si trova nell'**inserimento dei dati**. Se da una parte per i database NoSQL l'inserimento dei dati risulta più facile e privo di rischi, per i database relazionali un cattivo inserimento può portare alla corruzione dei legami tra tabelle, e quindi all'ottenimento di dati poco edificabili.

Integrità dei dati

Con un database NoSQL è possibile costruire un documento adatto alla mappatura delle classi-oggetto del proprio applicativo riducendo di molto i tempi di sviluppo. La *mancanza dei controlli fondamentali sull'integrità dei dati*, tuttavia, delega all'applicativo che dialoga con il database questo compito, che ovviamente dovrà essere testato in maniera più approfondita prima di essere messo in produzione. Nei database relazionali questo tipo di controllo avviene all'interno dello stesso database.

1.2.5 MySQL

Quotidianamente, immense quantità di informazioni vengono affidate a tecnologie che ne garantiscono la conservazione duratura ed un recupero efficiente che ne permetta l'analisi. Da anni, questo ruolo viene interpretato molto bene da un prodotto software completo, efficiente ed affidabile: MySQL [5] . Questo software è un DBMS (database management system) open source per la gestione di database relazionali e rappresenta una delle tecnologie più note e diffuse nel mondo dell'IT. I programmi che devono interagire quindi con una base di dati non possono farlo direttamente, ma devono dialogare con il DBMS, che è l'unico ad accedere fisicamente alle informazioni.

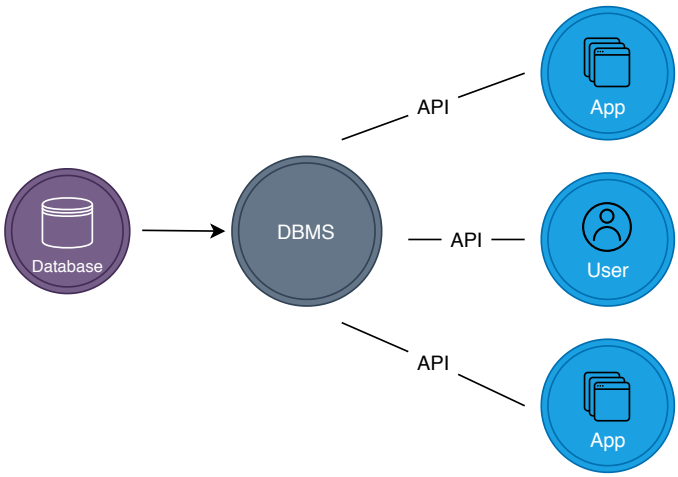

Figura 1.8. Database management system

Oltre alla gestione del database, un DBMS deve controllarne l'accesso concorrente, assicurarne la sicurezza e l'integrità dei dati e permetterne la condivisione e l'integrazioni con applicazioni differenti. Grazie a queste caratteristiche le applicazioni che vengono sviluppate possono contare su una sorgente dati sicura, affidabile e generalmente scalabile.

Il linguaggio SQL

Il nome SQL deriva l'abbreviazione di *Structured Query Language*, un linguaggio di programmazione che consente di accedere e gestire i dati in un database relazionale. Questo linguaggio rappresenta lo **standard per database basati sul modello relazionale**. La diffusione di SQL è dovuta in buona parte alla intensa opera di standardizzazione dedicata a questo linguaggio, svolta principalmente nell'ambito degli organismi ANSI (*American national Standards Institute* [6], l'organismo nazionale statunitense degli standard) e ISO (l'organismo internazionale che coordina i vari organismi nazionali) [7].

Funzionalità e Sintassi A seconda dell'operazione che si vuole eseguire sul database, SQL mette a disposizione diverse tipologie di linguaggi:

- *DDL - Data Definition Language* per creare e modificare *schemi di database*
- *DML - Data Manipulation Language* inserire, modificare e gestire dati memorizzati
- *DQL - Data Query Language* per interrogare i *dati memorizzati*
- *DCL - Data Control Language* creare e gestire strumenti di controllo e accesso ai dati

Sarebbe quindi diminutivo definire SQL 'un semplice linguaggio di interrogazione' in quanto alcuni dei suoi sottoinsiemi sopra elencati permettono di creare, gestire e modificare il database. La popolarità di SQL è inoltre dovuta alla sua facilità di comprensione, dovuta a un linguaggio con comandi semplici, autoesplicativi nella loro sintassi, e prestante per qualsiasi tipo di operazione.

```
SELECT Matricola, Cognome, Nome, Corso, Voto
FROM Studenti JOIN Esami
ON Matricola = Studente
```

Matricola	Cognome	Nome	Corso	Voto
276545	Rossi	Maria	Analisi	28
485745	Neri	Fabio	Chimica	27

Figura 1.9. Esempio di Query SQL

1.3 Architettura di Microservizi

1.3.1 Architettura Monolitica

Ai primordi dello sviluppo applicativo, anche un cambiamento minimo a un software esistente imponeva un aggiornamento completo e un ciclo di controllo qualità. Le applicazioni erano sviluppate e distribuite come una singola entità, e tale approccio veniva spesso definito **"monolitico"**, perché il codice sorgente dell'intera applicazione era compilato in una singola unità di deployment.

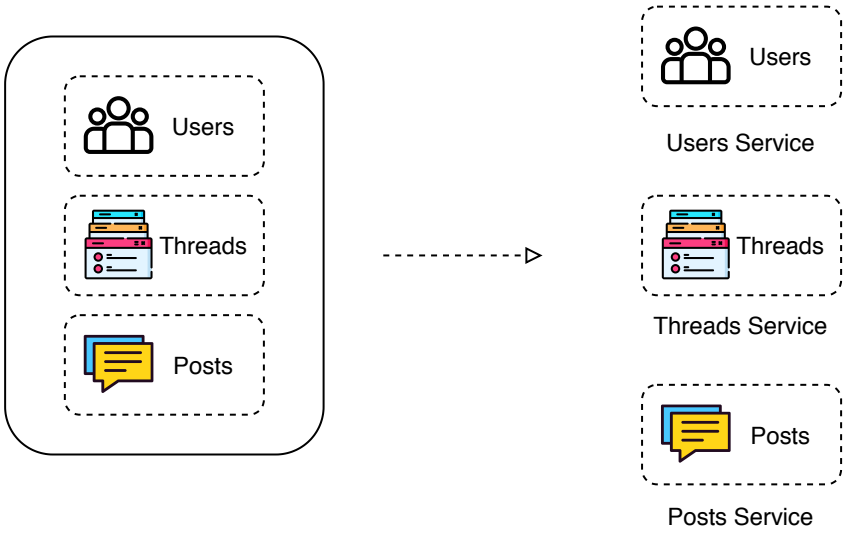

Figura 1.10. Architettura Monolitica e Microservizi

Le applicazioni che seguono questo tipo di architettura sono di facile implementazione, in quanto tipicamente raccolte all'interno di un unico progetto e distribuite in un unico pacchetto. Questo tipo di architettura si presta bene per applicazioni piccole o comunque poco soggette a cambiamenti, ma la cosa cambia quando ci troviamo a sviluppare applicazioni complesse che richiedono continui aggiornamenti. Se uno di questi dovesse causare errori, l'unica soluzione sarebbe quella di *disconnettere tutto* e fare un rollback totale del software: è chiaro che un'azienda non può permettersi tempi di inattività. L'unico modo di poter scalare un'applicazione monolitica è quello di replicare l'intera applicazione con conseguente aumento di costi e risorse necessarie. In seguito ai problemi derivanti da questo tipo di architettura, nacquero i primi studi di architetture a servizi, sulla base di uno dei dogmi dell'ingegneria del software:

Principio di Singola Responsabilità. *Riunire le cose che cambiano per lo stesso motivo e separare quelle che cambiano per motivi diversi.*

Cosa sono i Microservizi

L'architettura di microservizi rappresenta un'approccio all'avanguardia per lo sviluppo e l'organizzazione del software. Secondo questo stile, il software è composto da servizi indipendenti di piccole dimensioni che hanno come finalità lo svolgimento di un unico compito, e di farlo nel migliore dei modi [8]. Ciascun microservizio, indipendente dagli altri, è dunque gestito da un unico team di sviluppo. Per una definizione più precisa riprendiamo le parole di Martin Fowler [9], considerato uno dei massimi esperti in materia, che afferma: *«Lo stile architetturale a microservizi è un approccio allo sviluppo di una singola applicazione come insieme di piccoli servizi, ciascuno dei quali viene eseguito da un proprio processo e comunica con un meccanismo snello, spesso una HTTP API.»*

1.3.2 Le origini

Il primo tentativo di architettura a servizi nasce con l'`architettura SOA`. Il concetto delle *Service-Oriented Architecture* si afferma all'inizio degli anni Duemila come una collezione di servizi indipendenti che comunicano gli uni con gli altri tramite un *Enterprise Service Bus (ESB)*. L'architettura a microservizi è una chiara **evoluzione dell'architettura SOA**, spinta dall'esigenza di una sempre più' marcata scalabilità, la quale permette di reggere il carico di milioni di utenti connessi in un determinato istante.

1.3.3 Le differenze con SOA

Per studiare le differenze tra le due architettura facciamo riferimento alla figura [1.11]:

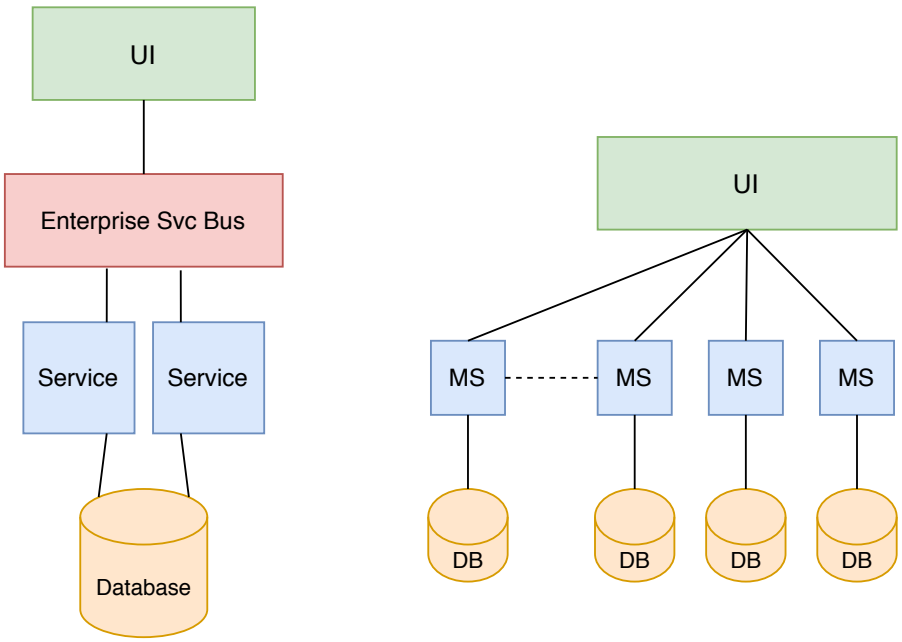

Figura 1.11. Architettura SOA e Microservizi

Granularità dei Servizi Il numero dei servizi è un buon fattore per la differenziazione delle due architetture. Lo stesso Martin Fowler, afferma che in un'architettura SOA non si arriva neanche ad una decina di servizi mentre in un architettura a microservice il numero dei servizi e' molto più alto: basti pensare che il servizio di streaming Netflix ha dichiarato di fare uso di oltre di 700 microservizi [10].

Comunicazione L'ESB può non essere presente in alcune architetture SOA, mentre i microservizi non ne prevedono l'utilizzo, preferendo comunicare direttamente tra loro con meccanismi di comunicazione light. Nella SOA, l'ESB potrebbe diventare un singolo punto di errore che influisce sull'intero sistema. Se per esempio ogni servizio comunicasse attraverso l'ESB, e uno di questi dovesse rallentare, potrebbe ostruire l'ESB con le proprie richieste da gestire. D'altra parte, i microservizi sono molto migliori nella tolleranza agli errori: se un microservizio presenta un errore di memoria, verrà interessato solo quel microservizio.

Database In SOA i servizi condividono gli storage mentre con i microservice ogni servizio può avere un database indipendente.

1.3.4 Caratteristiche dei Microservizi

Autonomia

Ciascun servizio nell'architettura basata su microservizi può essere sviluppato, distribuito, eseguito e ridimensionato senza influenzare il funzionamento degli altri componenti. I servizi non condividono alcun codice o implementazione con gli altri.

Specificità

Ciascun servizio è progettato per una serie di capacità e si concentra sulla risoluzione di un problema specifico. Se nel tempo si decide di rendere un servizio più complesso, il servizio può essere scomposto in servizi più piccoli.

Eterogeneità delle Tecnologie

Durante lo sviluppo di un microservizio si ha totale libertà di scelta nell'utilizzo delle tecnologie. Ciascuna tecnologia viene decisa esclusivamente in base allo scopo del microservizio, senza basarsi sulla sua interazione con gli altri. Un microservizio può fare affidamento su un modello di database relazionale mentre un servizio con cui comunica su uno di tipo non relazionale, così come la loro logica può essere scritta in linguaggi differenti.

Semplicità di Distribuzione

Con i microservizi è possibile apportare una modifica a un singolo servizio e distribuirlo indipendentemente dal resto del sistema con tecniche di continuos delivery del tutto automizzate. Questo permette di rilasciare aggiornamenti più velocemente e in maniera più sicura.

Resilienza

La resilienza è la capacità di accettare la possibilità' di errori e continuare a funzionare. Con i microservizi, le applicazioni possono gestire completamente gli errori di un servizio isolando la funzionalità senza bloccare l'intera applicazione.

Scalabilità

I microservizi consentono di scalare ciascun servizio in modo indipendente per rispondere alla richiesta delle funzionalità che un'applicazione supporta.

1.4 Servizi e API

1.4.1 Cos'è un'API

Un'*Application Program Interface (API)* può essere considerata come un *contratto* tra un fornitore di servizi e l'utente destinatario di tali dati: l'API si limita a stabilire il contenuto richiesto dal consumatore (la chiamata) e il contenuto richiesto dal produttore (la risposta). Un'API funge quindi da elemento di intermediazione tra gli utenti e le risorse che questi intendono ottenere. È anche un mezzo con il quale un'organizzazione può condividere risorse e informazioni assicurando al contempo sicurezza e controllo, poiché stabilisce i criteri di accesso. L'utilizzo di un'API permette all'utente di rimanere all'oscuro delle specifiche con cui le risorse vengono recuperate o della loro provenienza.

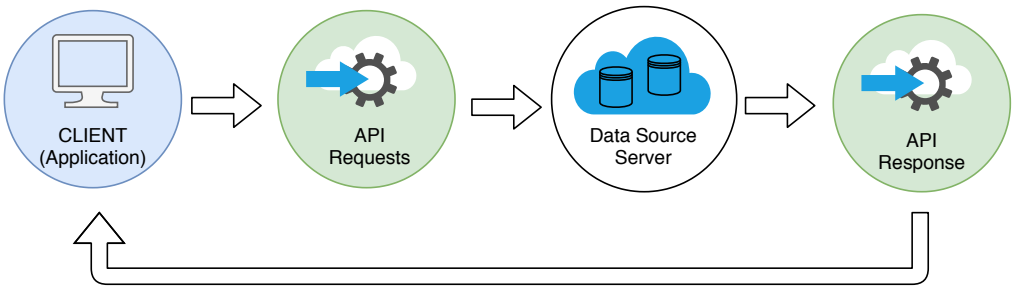

Figura 1.12. Funzionamento delle API

1.4.2 Representational State Transfer - REST

Il Representational State Transfer è lo stile architetturale più popolare degli ultimi anni. Non si tratta di un protocollo nè tantomeno di una specifica (in quanto non fa riferimento a uno standard ufficiale), ma piuttosto di una serie di vincoli per la creazione di un servizio web che può servirsi di standard, come HTTP, URI, JSON e XML. REST permette di *accedere* e *modificare* la **rappresentazione di risorse**, attraverso una serie di **operazioni stateless** *uniformi* e *predefinite*. Generalmente basato su HTTP, permette di utilizzare anche altri protocolli di trasferimento come SNMP, SMTP, ecc...

Perchè è conveniente? L'utilizzo di API REST fornisce una notevole quantità di libertà e flessibilità agli sviluppatori. Questo stile architetturale non richiede l'installazione di alcuna libreria (fatta eccezione per HTTP Client-Server e il JSON parser). Non vincola chi lo sceglie a utilizzare una tecnologia, ma al contrario ne supporta diverse.

Cosa la rende semplice? L'informazione, o rappresentazione, viene consegnata in uno dei diversi formati tramite HTTP: JSON (Javascript Object Notation), HTML, XLT o testo semplice. Il formato JSON è uno dei più diffusi, perché indipendente dal linguaggio e facilmente leggibile da persone e macchine.

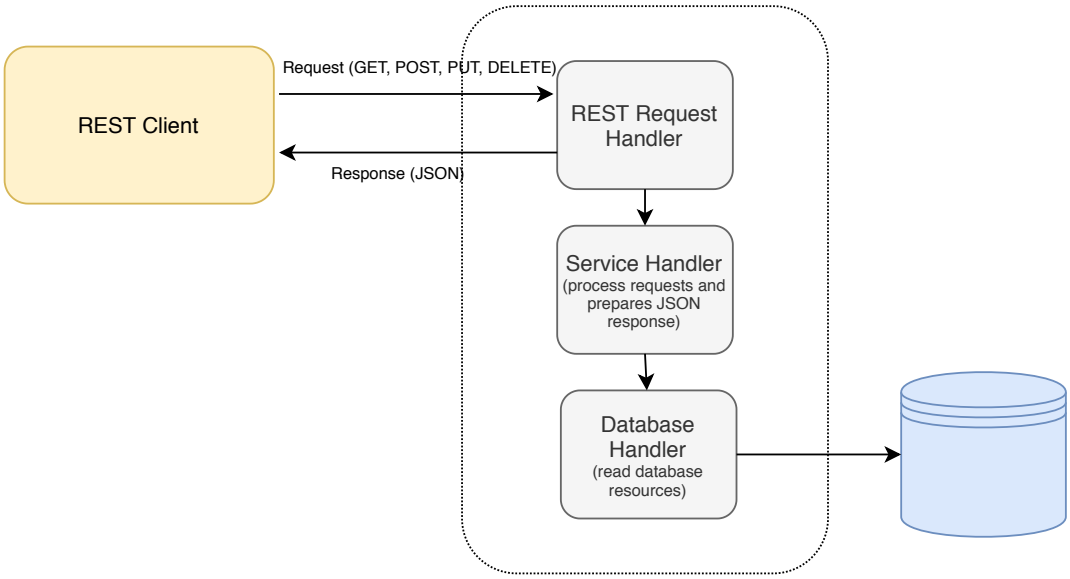

Figura 1.13. Funzionamento delle API REST

```
GET /messages HTTP/1.1
Host: domainname.com
Content-Type: text/plain;charset:UTF-8
Content-Length: 44
```

Figura 1.14. Esempio di Richiesta REST

Principi REST

Di seguito esponiamo i *sei principi per un'architettura REST*, per la prima volta introdotti durante la dissertazione nella tesi di dottorato di Roy Fielding, uno dei principali autori delle specifiche dell'Hypertext Transfer Protocol (HTTP). [11]

1. Client-Server Secondo questo principio, nello sviluppo di un'architettura REST Client e Server devono rimanere separati, in modo da potersi evolvere individualmente.

Per questo motivo, si dice che rispetta il paradigma dell'informatica noto come **Separation of Concerns (SoC)**, traducibile in italiano come *suddivisione dei compiti*. Secondo il principio SoC il sistema deve essere diviso in moduli distinti, ciascuno dedito allo svolgimento di un proprio compito, supportando in tal modo l'evoluzione indipendente della logica lato client e della logica lato server. Secondo questo vincolo il server deve offrire una o più funzionalità e ascoltare le richieste di possibili client. Un client deve invocare il servizio messo a disposizione dal server inviando il corrispondente messaggio di richiesta. Il servizio lato server a questo punto respinge la richiesta o esegue l'attività richiesta prima di inviare un messaggio di risposta al client. La gestione delle eccezioni è delegata al client.

2. Stateless Con questo termine si fa riferimento al tipo di operazioni. Le singole invocazioni delle API Rest non devono fare affidamento alle risorse presenti sul server, ma esclusivamente sui dati forniti nella stessa richiesta. Nel client non è previsto un sistema di memorizzazione delle informazioni delle richieste, ciascuna di queste è distinta e non connessa. Ciò garantisce una forte scalabilità, riducendo l'utilizzo di memoria sul server.

3. Cache Le Rest API devono essere progettate in modo da favorire l'utilizzo di dati cachable. *La richiesta di rete più efficiente è quella che non utilizza la rete.* In un'architettura REST i messaggi di risposta dal servizio ai suoi consumatori sono esplicitamente etichettati come memorizzabili nella cache oppure non memorizzabili. In questo modo, il servizio, il consumatore o uno dei componenti intermediari possono memorizzare nella cache la risposta per il riutilizzo nelle richieste successive. Le richieste vengono passate attraverso un componente cache, che può riutilizzare le risposte precedenti per eliminare parzialmente o completamente alcune interazioni sulla rete.

4. Interfaccia Uniforme Il Client deve essere in grado di comunicare con il server usando un singolo linguaggio, indipendentemente dall'architettura di backend. Questo permette alle informazioni di essere trasferite in una forma standard.

- **Identificazione delle Risorse** Una risorsa è un oggetto o la rappresentazione di qualcosa di significativo nel dominio applicativo. Il concetto di risorsa è quindi molto simile a quello di oggetto nel mondo della programmazione ad oggetti.
- **Manipolazione delle risorse attraverso rappresentazioni** Una risorsa può essere rappresentata in molti modi diversi. Ad esempio come HTML, XML, JSON o anche come file JPEG. I client interagiscono con le risorse tramite le loro rappresentazioni, il che è un modo potente per mantenere i concetti di risorse astratti dalle loro interazioni.
- **Hypermedia come motore dello stato dell'applicazione** L'applicazione deve essere guidata da collegamenti, consentendo ai clienti di scoprire risorse tramite collegamenti ipertestuali.

5. Sistema a strati In un sistema a livelli, componenti intermedi come i proxy possono essere collocati tra client e server. Uno dei vantaggi di un sistema a più livelli è che gli intermediari possono intercettare il traffico client-server per scopi specifici; ad esempio il caching, la sicurezza o l'autenticazione. Una soluzione basata su REST può essere composta da più livelli architettonici indipendenti l'uno dall'altro.

6. Codice su richiesta Vincolo opzionale. Richiede che il codice eseguibile possa essere trasmesso su richiesta creando un'applicazione che non dipende più esclusivamente dalla propria architettura. I problemi di sicurezza e l'eterogeneità dei linguaggi di programmazione tuttavia costituiscono un limite per l'adozione di questo vincolo.

Operazioni CRUD

Con il termine si fa riferimento all'acronimo delle operazioni mappate ai metodi HTTP:

- *CREATE* → *POST*
- *READ* → *GET*
- *UPDATE* → *PUT*
- *DELETE*

Ad ogni tipo di operazione sul database corrisponde quindi un metodo HTTP. L'url di una risorsa la identifica univocamente, mentre il metodo della chiamata definisce l'operazione che si va a fare su di essa.

Task	Method	Path
Create a new task	POST	/tasks
Delete an existing task	DELETE	/tasks/{id}
Search for task	GET	/tasks
Get a specific task	GET	/tasks/{id}
Update an existing task	PUT	/tasks/{id}

Figura 1.15. Esempio di Operazioni CRUD

L'unica operazione che ha un comportamento particolare è il *login*. Nonostante si tratti di un controllo delle credenziali, e quindi di un'operazione di lettura di dati sul database, al *login corrisponde un metodo POST*. Questo è dovuto al fatto che passando i parametri di autenticazione attraverso una GET, questi sono passati in chiaro e visibili nell'url della richiesta. Utilizzando il metodo POST è possibile nascondere dei dati più sensibili.

1.4.3 Web Services

Cos'è un Web Service

Secondo la definizione data dal World Wide Web Consortium (W3C), si definisce *Web Service* un *sistema software* che si mette al servizio di un'applicazione ed è progettato per

supportare l'interoperabilità tra diversi elaboratori sulla medesima rete [12]. Il servizio web è in grado di offrire un'*interfaccia software* comprendente un file WSDL (Web Services Description Language) che descrive il servizio in modo più dettagliato. Il client può utilizzare il file WSDL per capire quali funzioni può svolgere sul server tramite le operazioni esposte dall'interfaccia, e in che modo le richieste devono essere costruite. Infine, la comunicazione funziona attraverso diversi protocolli e architetture.

Simple Object Access Protocol - SOAP

SOAP è un protocollo ufficiale gestito dal W3C (World Wide Web Consortium), ideato inizialmente per consentire la comunicazione tra applicazioni realizzate con linguaggi e piattaforme diverse. È un protocollo per la comunicazione client-server e viene utilizzato con lo standard WSDL. Trattandosi di un protocollo, richiede l'integrazione di regole che ne aumentano la complessità e il carico di gestione sul sistema, comportando tempi di caricamento delle pagine più lunghi. Oltre alle regole, integra anche standard di conformità che lo rendono idoneo agli ambienti enterprise. Oltre alla sicurezza, gli standard di conformità integrati includono:

- *Atomicità*
- *Coerenza*
- *Isolamento*
- *Durata*

Dal loro acronimo l'insieme di questi standard prende il nome di **ACID**.

Richieste e Risposte

Una richiesta di dati inviata a un'API SOAP può essere gestita tramite uno dei protocolli a livello applicativo: HTTP (per i browser web), SMTP (per l'email), TCP e altri. Una volta ricevuta la richiesta, i messaggi SOAP di ritorno devono essere restituiti come documenti in formato **XML**, un linguaggio di markup facilmente leggibile da utenti e macchine. Una *richiesta completata a un'API SOAP non viene memorizzata nella cache del browser*, e pertanto non sarà possibile accedervi successivamente senza rinviarla all'API.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <getProductDetails xmlns="http://domainname.com/ws">
        <productId>827635</productId>   <!-- ID della risorsa -->
        </getProductDetails>
    </soap:Body>
</soap:Envelope>
```

Figura 1.16. Esempio di Richiesta SOAP

Differenze con REST

La differenza sostanziale tra le due tecnologie introdotto risiede nel fatto che SOAP è un protocollo con requisiti specifici, contrariamente a REST che è un insieme di linee guida con implementazione flessibile. Molti sistemi esistenti aderiscono ancora a SOAP, mentre REST, il cui avvento è successivo, è spesso considerato come un'alternativa più rapida negli scenari web. Poiché sono ottimizzate, le API REST sono adatte ai contesti più innovativi come l'Internet of Things (IoT), lo sviluppo di applicazioni mobili e il serverless computing. Grazie all'integrazione della sicurezza e della conformità delle transazioni, i servizi web SOAP rispondono a molte esigenze aziendali, ma risultano anche più complessi.

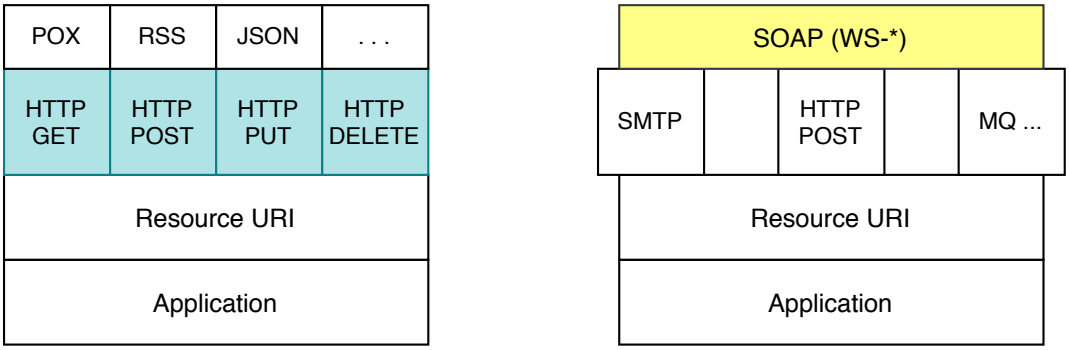

Figura 1.17. REST vs. SOAP

Un servizio SOAP, indirizza le richieste sempre verso un unico indirizzo chiamato *end-point*. Tutte le richieste vengono effetuate sempre con il metodo POST e questo porta ad un maggior traffico verso l'end-point in quanto in tutte le richieste viene sempre inviato un documento XML. Nei servizi REST le richieste vengono indirizzate verso URI differenti che si mappano sulle risorse. Il consumo di banda che porta un servizio REST è ridotto al minimo, infatti viene inviato insieme alla richiesta un documento XML o altre informazioni oltre all'URI solo quando bisogna creare o aggiornare lo stato di una risorsa [1.17].

Capitolo 2

Architettura Funzionale del Sistema

In questo capito è illustrato il funzionamento attuale del sistema e le tecnologie che utilizza, con cenni riguardenti le soluzioni pensate per ogni specifico problema. Vengono presentate l'architettura del sistema di partenza e quella di arrivo, in riferimento al modello di sviluppo dell'*ingegneria del software* seguito.

2.1 Reverse Enginering

Definizione. *La reingegnerizzazione è quel processo di esame, analisi e alterazione di un sistema software esistente, col fine di ricostruirlo in una nuova forma, e la successiva implementazione. [13] Nel processo di reingegnerizzazione è compresa anche la riprogettazione del software, mentre quando si parla di reverse engineering si fa riferimento all'analisi.*

Il reverse engineering del codice permette di invertire i processi di sviluppo e produzione di un software e quindi di ottenere uno sguardo prezioso dietro le quinte di un programma. Attraverso lo studio dettagliato del codice sorgente è possibile comprendere, riscrivere o ricostruire l'architettura di un programma, il suo funzionamento e le sue strutture interne. Apparentemente la reingegnerizzazione non dovrebbe essere fatta per modificare i requisiti funzionali del sistema. In realtà, spesso quando si applica questa tecnica ci si accorge, in fase di riprogettazione, che alcuni requisiti, dopo anni di sviluppo, non sono più validi e andrebbero tolti dal disegno, mentre altri, nuovi, andrebbero introdotti.

2.1.1 I motivi

La pandemia di quest'ultimo anno ha portato l'azione di refactoring su *Zerocoda* ad essere a tutti gli effetti una necessità. Il distanziamento sociale è un bisogno per ogni tipo servizio offerto ad una clientela più o meno vasta, e tenere traccia dei clienti e del loro numero temporizzando gli accessi al servizio è senza alcun dubbio la soluzione migliore. Nello stato attuale, tuttavia, l'applicazione non rispetta i requisiti funzionali per essere rilasciata ad un pubblico di tipo *enterprise*. Sono sempre di più le aziende che hanno richiesto l'inserimento dei loro servizi nell'applicazione. L'inserimento di nuovi servizi porterebbe all'aumento

delle richieste da parte dei client, più richieste a più dati da elaborare, e più dati a un sovraccarico. L'applicazione non è pronta a uno scenario di questo tipo, pertanto urge un refactoring mirato all'introduzione di nuove tecnologie, che renda il software più scalabile in prevenzione del nuovo numero di clienti sulla piattaforma.

Questo rappresenta il motivo principale, quello che ha portato l'azione di refactoring, dapprima solo ipotizzata da coloro che già lavoravano al sistema, a un inizio di progettazione. Normalmente le esigenze possono essere anche altre:

- il team di sviluppo è cambiato e sono subentrati altri programmatori che non hanno conoscenze in merito al sistema;
- la documentazione è assente del tutto o, se presente, dopo anni di sviluppo e modifiche non è stata aggiornata e risulta obsoleta;
- l'introduzione di piccoli cambiamenti risulta complessa e costosa da affrontare in termini di tempo e risorse;
- il software è spesso soggetto a continue correzioni di bug, indice che la struttura è precaria e mal progettata;
- numerose correzioni sono state apportare e si è verificato il fenomeno "architectural drift" (deriva architetturale), cioè il sistema si è spostato molto dal disegno originale che non è più chiaro
- i metodi e gli strumenti di sviluppo originali sono ormai superati ed è difficile trovare programmatori con queste conoscenze.

2.2 Modello di Sviluppo

Come *metodologia di sviluppo* si è scelto di adottarne una di tipo **agile**. Secondo questo *principio teorico*, il punto di partenza è costituito dalla specifica dei requisiti, mentre lo sviluppo vero e proprio del software avviene solo in seguito.

2.2.1 Waterfall Model

Il modello a cascata è un **ciclo di vita lineare**, che suddivide il processo di sviluppo in fasi di progetto consecutive. A differenza dei modelli iterativi, *ogni fase viene eseguita solo una volta*. Con questo modello il progetto viene organizzato in una sequenza di fasi, ciascuna delle quali produce un output che costituisce l'input per la fase successiva.

Origini

Lo sviluppo del modello viene attribuito allo scienziato informatico Winston W. Royce, che tuttavia, non è stato il suo vero inventore. Al contrario, il suo articolo pubblicato nel 1970 "Managing the Development of Large Software Systems" [14] conteneva una critica aperta nei confronti dei cicli di vita lineari. Come alternativa, presentava un modello iterativo-incrementale, in cui ogni fase attinge a quella precedente e ne verifica i risultati.

Il modello di Royce è cosituito da sette fasi eseguite in diversi passaggi (iterazioni):

1. Requisiti di sistema
2. Requisiti di software
3. Analisi
4. Progettazione
5. Implementazione
6. Test
7. Esecuzione e Manutenzione

Il modello a cascata si ispira alle fasi definite da Royce, ma prevede solo un'iterazione.

Funzionamento

Nella pratica vengono utilizzate diverse versioni del modello a cascata. Attualmente sono in uso modelli che suddividono i processi di sviluppo in cinque fasi. Le fasi 1, 2 e 3, definite da Royce, sono riunite in un'unica fase di progetto, chiamata analisi dei requisiti. [2.1]

1. Analisi: pianificazione, analisi e specificazione dei requisiti
2. Progettazione: progettazione e specificazione del sistema
3. Implementazione: programmazione e test di modulo
4. Test: integrazione di sistema, test di sistema e test di integrazione
5. Esecuzione: rilascio, manutenzione, miglioramento

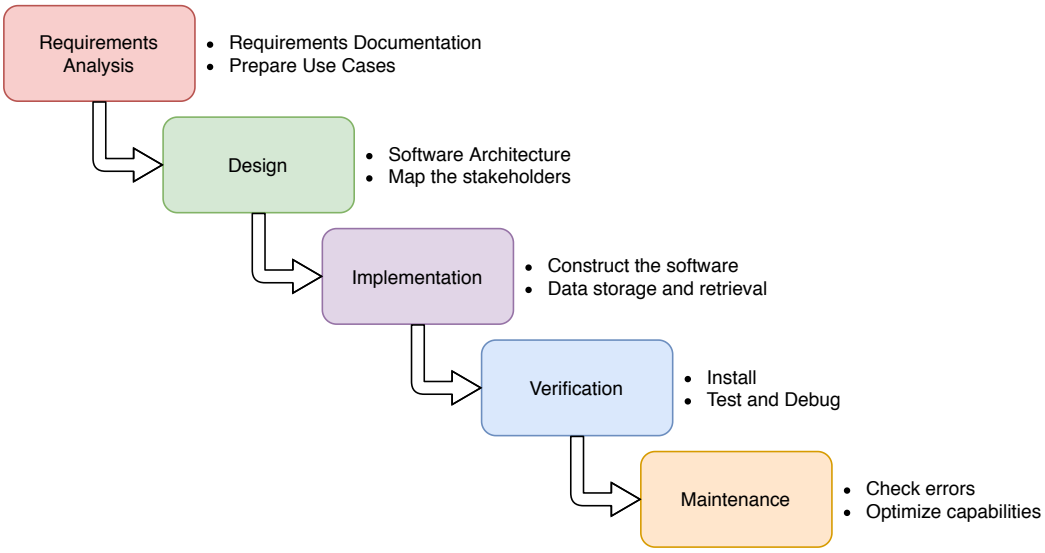

Figura 2.1. Fasi del Modello a Cascata

Il modello da noi utilizzato rappresenta un'*estensione del modello a cascata*, in quanto l'eseguire ogni fase una sola volta comporta un grande vincolo. Ad ogni fase infatti si confrontano e verificano i risultati ottenuti con quelli della fase precedente. Partendo da un'analisi delle esigenze richieste dall'applicazione, dai suoi limiti e dalle necessità dei clienti, si è poi passati all'individuazione di una strada per lo sviluppo.

2.3 Funzionamento dell'Applicazione

2.3.1 Scenario

Lo scopo di quest'applicazione è quello di *velocizzare l'accesso i servizi*. Senza questo *primo requisito*, l'applicazione non troverebbe posto nel mondo enterprise. Per questo motivo, l'interfaccia utente risulta semplice, e comprende poche funzionalità.

L'utente per accedere ai servizi deve registrarsi al sistema fornendo i propri dati personali (che potranno essere modificati in qualsiasi momento), scegliendo un'email e una password per le future autenticazioni. Anche senza essere registrato può cercare una struttura tra quelle presenti e scegliere un servizio che questa eroga.

I servizi possono essere diversi: analisi, esami, visite mediche, qualsiasi servizio che preveda un'attesa del cliente potrebbe essere inserito da una struttura sanitaria. Selezionando un servizio, all'utente viene mostrato il relativo calendario di disponibilità, in termini di giorni e fasce orarie.

La prenotazione avviene una volta selezionato un orario (corrispondente a uno *slot*) e confermata l'intenzione di volerlo riservare. Per fare quest'operazione è necessario che l'utente sia autenticato al sistema. Prima di confermare la prenotazione di uno slot, l'utente può scegliere se inserire i dati (almeno uno tra nome e cognome, email o telefono) di un altro utente, anche non iscritto al sistema, per cui prenotare il servizio. Scegliendo questa opzione il destinatario sarà informato mediante delle notifiche sui futuri eventi relativi allo stato della sua prenotazione. A questo punto all'utente viene fornito un "biglietto virtuale".

Il biglietto consiste in una numerazione. Questa numerazione corrisponde al numero che verrà chiamato allo sportello (si pensi ad un biglietto virtuale staccato al momento della prenotazione). Ciascun servizio è identificato da una lettera dell'alfabeto (A, B, C, ...) mentre ciascun numero identifica il cliente in coda (1, 2, 3, ...). In questo modo la numerazione B12, ad esempio, fa riferimento al cliente numero 12 di un servizio, C13 ad un altro cliente di un altro servizio. L'idea è che chi ha prenotato acceda alla struttura nel momento esatto in cui viene erogato il servizio, evitando coda e assembramenti.

La struttura , così come l'utente, deve avere informazione di questo biglietto virtuale e questo avviene attraverso un *software* installato in loco che prende il nome di **mru**.

L'MRU è un software che disponde di un'interfaccia web per gli operatori e per i totem che vengono utilizzati. Tutte le notti l'MRU interroga Zerocoda richiedendo le prenotazioni per la giornata odierna. Se ci sono le scarica, in questo modo anche gli operatori hanno informazione delle prenotazioni.

2.3.2 Use Case Diagram

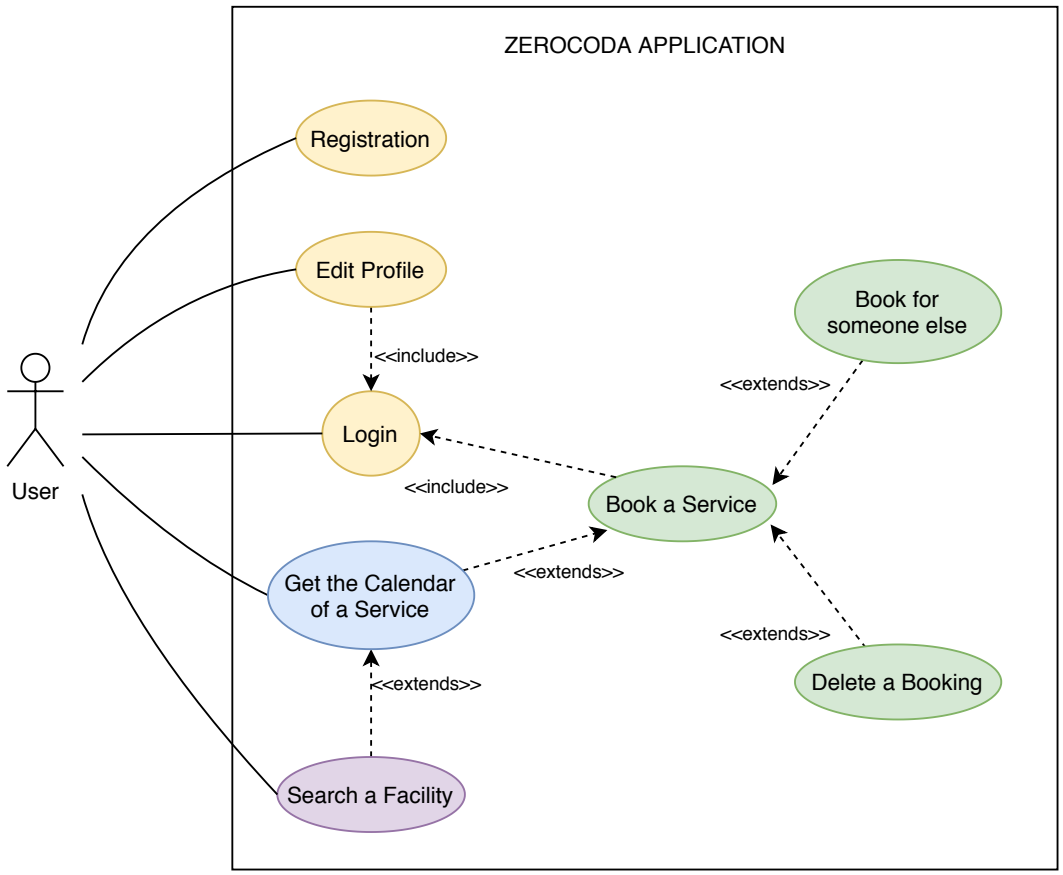

Figura 2.2. Zerocoda Use Case Diagram

In figura [2.2] è presentato il diagramma dei casi d'uso di un utente generico di Zerocoda. I diversi colori utilizzati per le operazioni che può fare rappresentano la *divisione in servizi* scelta per la fase di implementazione delle REST Api.

2.3.3 I diversi Enti Zerocoda

Sull'applicazione Zerocoda è possibile prenotare i servizi erogati da diversi enti. Alcuni di questi, tuttavia, non sono presenti sul sito. Ci sono strutture sanitarie che non vogliono che i loro utenti accedano a un sito generalista come *Zerocoda.it*. Questi vogliono che il loro sito funga da "vetrina", che l'utente riconosca la struttura presso cui prenota attraverso

elementi come il logo, il nome, ecc... L'utilizzo del **virtual hosting** risponde a questa esigenza. In fase di accesso il servizio compare come se fosse erogato dell'ente privato. Questi *siti premium* consistono in realtà in un sito statico (come lo stesso Zerocoda), ma con una configurazione dedicata, che una volta ricevuta dal browser cambia il *look and feel*.

É possibile offrire una personalizzazione maggiore ai siti che la richiedono. La richiesta può spaziare dall'utilizzo di colori particolari per gli elementi della pagina: quali pulsanti, form, colori del background, a quella di immagini e loghi personalizzati o alla necessità di avere funzionalità esclusive rispetto ad altre strutture.

La configurazione consiste in un file JSON contenente tutte le informazioni del sito. Oltre alle informazioni relative al nome, indirizzo, descrizione, questo contiene anche stringhe di testo già formattate in HTML, da inserire all'interno della pagina in appositi spazi. Può contenere anche informazioni riguardanti lo stile (CSS) della pagina. La configurazione viene caricata per ultima, dopo che i componenti statici sono stati renderizzati. Viene inviata al frontend attraverso uno script JavaScript.

2.4 Analisi dell'Applicazione

In numeri , i clienti (enti sanitari) per Zerocoda sono 45. Il numero comprende sia gli enti presenti su Zerocoda, sia coloro che hanno richiesto una personalizzazione su un proprio portale. Il sistema, in totale vanta i seguenti numeri:

- 1 Zerocoda
- 17 siti con configurazione personalizzata
- 500.000 utenti registrati
- 298 impianti di MRU registrati

2.4.1 Multitenancy

La particolarità del sistema è quella di essere **multitenant** [2.3]. Con un solo servizio è possibile offrire esperienze isolate agli utenti: una diversa esperienza sulla base della loro organizzazione. In questo modo, se un nuovo ente richiedesse che tutti i suoi dati fossero salvati su un nuovo database, si potrebbe installare un nuovo ambiente separato apposito. In alcuni casi è possibile anche chiedere metodi di registrazione o accesso al sistema ad-hoc, evitando il sistema standard basato su email e password. Così facendo, il sistema si limiterebbe ad esporre un'API in aggiunta all'applicazione condivisa da tutti, che gestisce la richiesta di autenticazione personalizzata sulle necessità dell'ente richiedente.

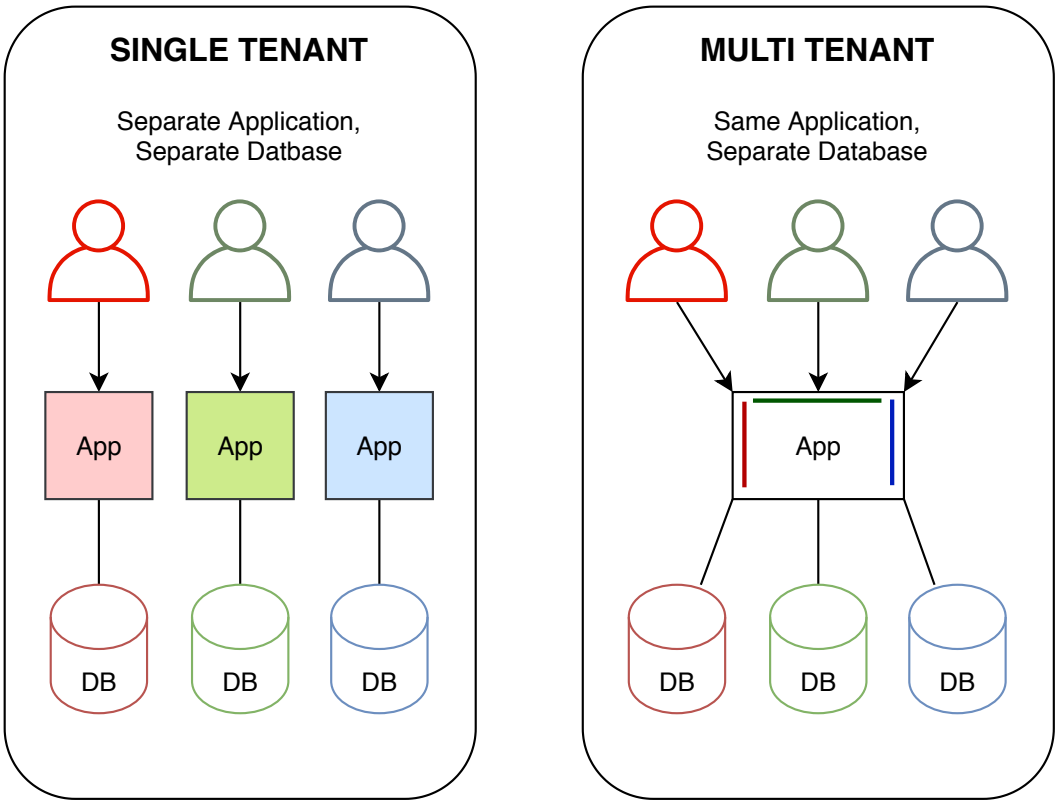

Figura 2.3. Single Tenant vs. Multi Tenant

2.4.2 Backend Overview

Ad oggi non è presente un elemento definibile come *layer di API* nell'applicazione. Il sistema è composto da un unico monolite che tra le tante mansioni che ha espone anche le 'API'. Nello stesso sistema viene effettuato ogni genere di controllo per quanto rigurarda l'autenticazione e i parametri passati nelle richieste. Il backend è interamente scritto in PHP e si appoggia ad un database MySQL. Nonostante con gli ultimi aggiornamenti PHP offra la possibilità di programmare a oggetti, questo non è il caso del sistema preso in analisi. Per la creazione di dati che in altri linguaggi sarebbero rappresentabili attraverso oggetti, il backend utilizza un sistema di liste con elementi di tipo chiave-valore.

2.4.3 Chiamate delle API

Le chiamate dal frontend alle API intercettano il click sul documento, andando a cercare il componente su cui è stata registrata l'interazione. Tutte le richieste inviate al backend sono richieste GET HTTP. Quindi, passando un qualsiasi parametro o effettuando operazioni di aggiunta o modifica di dati, i parametri vengono aggiunti all'url, e mostrati in chiaro. Questo scelta è stata adottata anche per i metodi di registrazione e autenticazione, i più delicati dal punto di vita della sicurezza. Con la sintassi `$_GET` in PHP si fa riferimento a una variabile globale che contiene i parametri in GET della richiesta HTTP.

```
http://domain.it/index.php/api/v1/login?

_apikey=1&
email=prova123%40gmail.com&
password=Prova123!
```

Figura 2.4. Esempio di richiesta GET in HTTP

Prendendo in analisi la richiesta di login in figura [2.4], nell'url sono presenti i parametri *email* e *password* utilizzati per autenticare l'utente. Questi dati sensibili *non dovrebbero appartenere all'url* dove sono visibili, ma andrebbero passati nel *body* di una richiesta. Con la variabile `$_GET` in PHP è dunque possibile accedere anche a questi parametri.

Scenario

Il sistema, per capire quale API è stata chiamata, ricerca il metodo della richiesta passando il parametro `$_GET` all'interno di un costrutto condizionale di grandi dimensioni e verificando ogni chiamata possibile. In figura [2.5] viene presentata una ricostruzione di parte del codice a cui si fa riferimento. Quando il valore del parametro corrisponde al nome del metodo chiamato, nello stesso ciclo richiama l'operazione richiesta. Risulta facile notare che il codice, implementato in questo modo, presenta diversi lati negativi. Vengono ripetute parti di codice, i parametri vengono istanziati dentro ciascun *if statement* e passati ai metodi che ne richiedono l'utilizzo: tutto questo all'interno della stessa condizione. Infine, i risultati di ciascuna chiamata vengono istanziati all'interno di array.

```
if ($_GET['id'] == "getReservationsDays") {
    $keyname =  ...
    $sys_id = ...
    ...
    $resultArr = $this->execAvailableReservationsDays($keyname, $sys_id);
    $this->responseAsJson($resultArr);
} else if ($_GET['id'] == "listEnterprises") {
    $sys_id = ...
    $origins = ...
    ...
    $resultArr = $service->getListDetailsEnterprises($sys_id, $origins);
} else if {
    ...
} ...
```

Figura 2.5. Struttura delle chiamate Api

2.4.4 Same Origin Policy

Un'origine è definita da un *protocollo*, un *dominio* e una *porta* di un URL. Più in generale, i documenti appartenenti a pagine web differenti sono isolati gli uni dagli altri. L'intento di questa politica è quello di consentire l'accesso a siti non sicuri, ma senza che quest'ultimi abbiano la possibilià di interferire con la sessione di chi naviga su un sito sicuro. [15]

I controlli di origine vengono applicati dal browser in ogni caso di potenziale interazione tra elementi di origini diverse. Questo include, ma non è limitato a:

- Codice JavaScript e Document Object Model (DOM)
- Cookies
- Chiamate AJAX (XmlHTTPRequest)

Chiamate AJAX

L'*Asynchronous JavaScript and XML* è ad oggi la tecnica più utilizzata per sviluppare applicazioni web interattive. Il concetto che sta alla base di una chiamata AJAX è quello di *poter scambiare dati tra client e server senza ricaricare la pagina*: lo scambio avviene in background tramite una chiamata asincrona dei dati di solito utilizzando l'oggetto *XMLHttpRequest*. Il framework **JQuery** semplifica notevolmente l'implementazione di chiamate di questo tipo, e a ciò deve la sua fama.

Cross Origin Resource Sharing - CORS

In alcuni casi, lavorare con domini differenti che interagiscono tra loro può rivelarsi *una necessità*. Quando ciò accade, *è possibile allentare la Same Origin Policy* in modo che non ostacoli le funzionalità di interazione tra domini dell'applicazione web. Ciò può essere fatto in diversi modi, ad esempio dichiarando l'origine utilizzando JavaScript, l'intestazione di una chiamata, o stabilendo un sistema di autenticazione tra le due parti. Questa procedura prende il nome di *Cross-Origin Resource Sharing*. L'utilizzo di questo meccanismo può fare a caso nostro per quanto riguarda le chiamate delle API di Zerocoda (che si trovano su un dominio differente rispetto a quello della pagina web), tuttavia in fase di sviluppo non era stato pensato, e i precedenti sviluppatori hanno elaborato un modo per aggirare il problema, piuttosto che eliminarlo alla radice. In fase di analisi si è tenuto conto di questo aspetto, pertanto durante l'implementazione si lavorerà alla stesura di API su un differente dominio, garantendone l'accesso e dichiarando affidabili le origini necessarie.

2.5 Architettura del Sistema

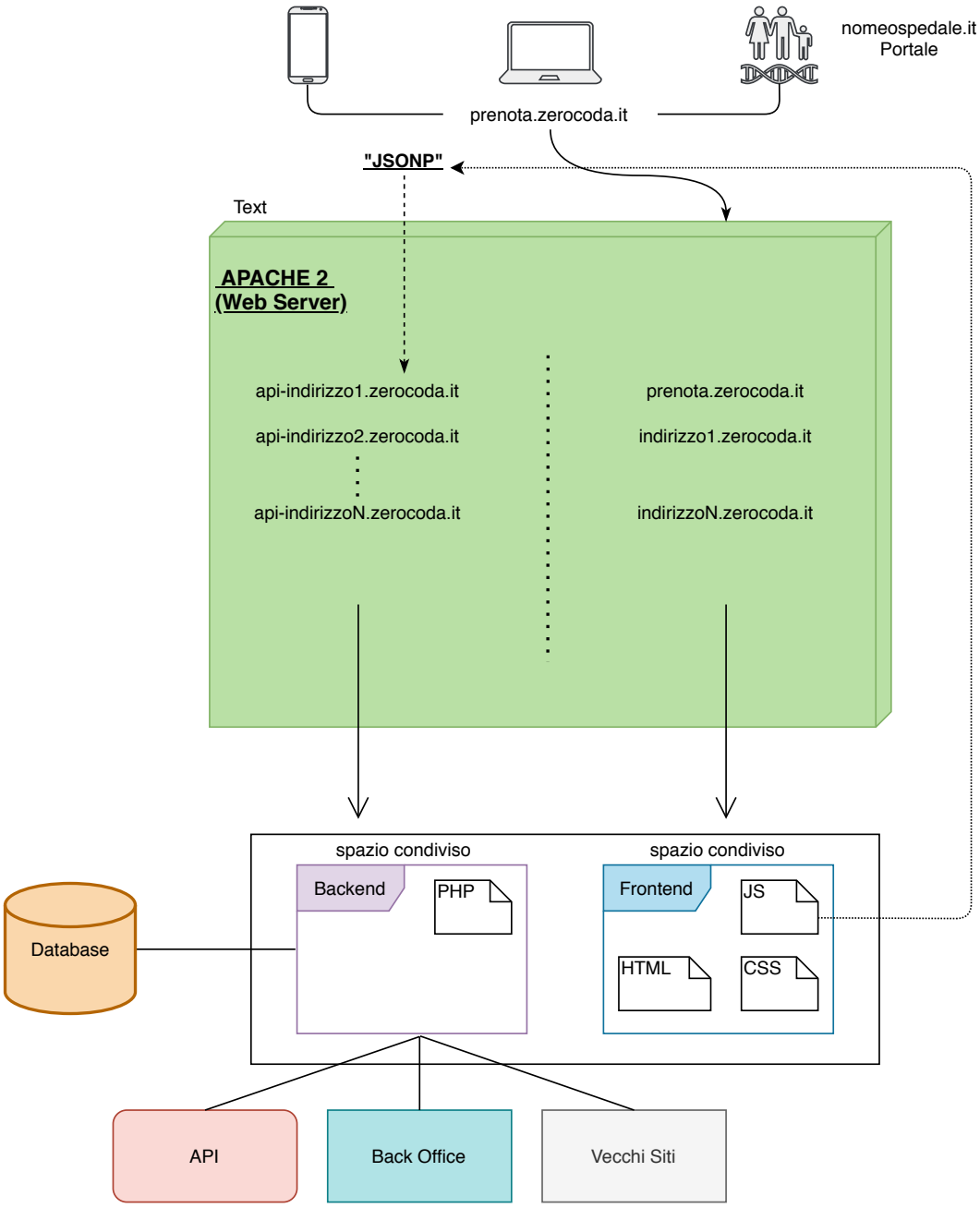

Figura 2.6. Architettura del Sistema di Partenza

2.5.1 Web Server Apache

A livello di file system si ha uno spazio condiviso per il backend (interamente in PHP) e uno spazio condiviso per il frontend, contenente i soli contenuti statici (HTML, CSS e immagini). Il backend funge da colonna portante per diversi componenti:

- **backoffice**: sostanzialmente l'interfaccia amministrativa di Zerocoda. Attraverso questa interfaccia è possibile gestire tutto il sistema. Viene utilizzata per aggiungere quelli che si definiscono 'slot'. Uno slot non è altro che un posto prenotabile per un servizio in un determinato giorno e in una determinata fascia oraria. Con il backoffice un amministratore può aggiungere in modo semi-automatico gli slot per un determinato servizio, scegliendone il numero per ogni ora e lo stato (se abilitato o disabilitato) al momento della creazione.

- **vecchi siti** precedenti al refactoring. Prima di questo refactoring il frontend del sito era stato aggiornato. Nulla è stato cambiato per quanto riguarda il funzionamento, solo gli elementi grafici sono stati aggiornati per rispondere ad uno stile un po' più moderno. Nonostante ciò, si è scelto di mantenere online il frontend per alcuni siti, che in questo modo rimane sempre a carico di Apache.

- **API**: le chiamate esposte in precedenza. Le Api a cui si fa riferimento quando si invia una richiesta per effettuare un'operazione sono le stesse per tutti i dispositivi. Non c'è differenza che si facciano da un browser web, un dispositivo mobile, o dal portale di un sito con configurazione personalizzata.

Con le caratteristiche attuali Apache2 funziona esclusivamente da Web Server.

2.5.2 Virtual Hosting

Non tutti i siti hanno abbastanza traffico da giustificare il costo di un web server dedicato, pertanto la loro condivisione su un singolo web server può essere una buona soluzione per abbassarne il costo. Il *virtual hosting* viene incontro a questa esigenza, permettendo ad un unico server web di "ospitare" più siti (e/o applicazioni) web, i quali condivideranno, secondo opportune politiche di gestione, le risorse di elaborazione del server stesso. Consolidare più servizi su un'unica macchina è una pratica comunemente adottata poiché permette di ottimizzare l'utilizzo delle risorse hardware disponibili. Senza il virtual hosting, si renderebbe necessario attivare una nuova macchina server per ogni nuovo sito o applicazione web, con un conseguente aumento dei costi ed un possibile sottoutilizzo delle risorse hardware. Con l'attuale architettura *Apache ospita al suo interno una serie di virtual host.*

Funzionamento Il server Apache, con un unico indirizzo IP, ospita più di un nome di dominio. I diversi domini condividono la stessa porta, ma si può anche assegnare a ciascun dominio una porta specifica.

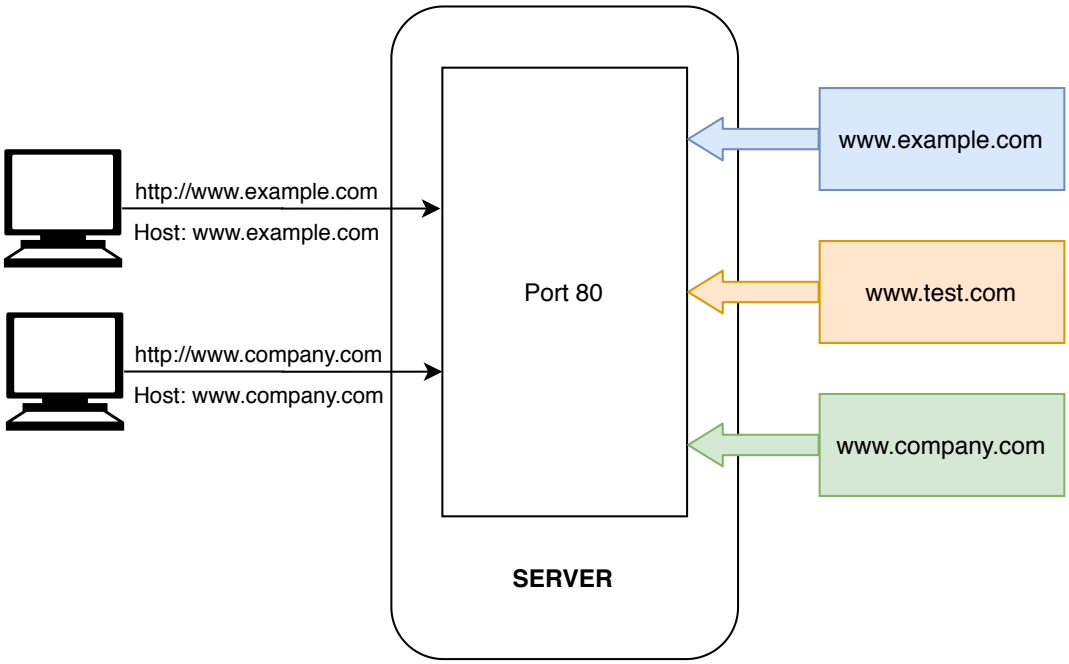

Figura 2.7. Funzionamento del Virtual Hosting

2.5.3 Database MySQL

Il dialogo con il database avviene attraverso il codice PHP del backend. E' compito del server Apache fornirne l'accesso per una serie di indirizzi (API). Il database è unico per tutti gli enti Zerocoda, anche gli enti 'premium' interrogano la stessa base di dati degli altri. Coloro che hanno richiesto un metodo di accesso diverso da quello standard hanno una tabella isolata dagli altri, contenente i dati dei propri utenti. Poiché questa tipologia di enti ha gli utenti su una tabella diversa, avrà anche API apposite per interrogarla. Questa API si trova sempre all'interno dello stesso backend, insieme alle altre.

2.6 Limiti del Database

La struttura del database non corrisponde a quella che ci si aspetterebbe in un tipico sistema di prenotazioni utente. Nel database sono infatti contenute diverse tabelle di verifica per il sistema. Non è presente nessuna struttura intermediaria tra client e server che si occupi di fare i controlli necessari per la verifica delle api keys e dei token utente, pertanto queste operazioni si traducono in delle semplici query alla base di dati. Inoltre, sul database vengono salvati anche i log riguardanti il funzionamento dello stesso: è logico dedurre che, in caso di malfunzionamento, non c'è modo di verificare questi documenti.

Oltre a quelle di verifica, il database presenta anche diverse tabelle di configurazione. Queste vengono per lo più utilizzate in fase di amministrazione dal backoffice.

Bibliografia

[1] Sergio Polini. «Introduzione alla Programmazione Orientata all'Oggetto». In: *MCmicrocomputer* 101-102.272-275,241-245 (1990). URL: https://issuu.com/adpware/docs/mc101/272.

[2] Paul R. Reed. *Developing Applications with Java and UML*. Addison-Wesley Professional, 2008. ISBN: 0201702525. URL: https://books.google.it/books/about/Developing_Applications_with_Java_and_UM.html?id=y9iCgb7rBoAC&redir_esc=y.

[3] Paul Jansen. «TIOBE Index for 2019». In: *TIOBE* (dic. 2019). URL: https://www.tiobe.com/tiobe-index/.

[4] Paolo Atzeni. *Basi di Dati*. McGraw-Hill International Education. McGraw-Hill, 2018, pp. 1–2, 15, 91, 108–109. ISBN: 9788838694455. URL: https://www.mheducation.it/informatica/informatica/basi-di-dati.

[5] *MySQL Database Service (IT)*. URL: https://www.mysql.com/it/.

[6] *American national Standard instute*. URL: https://www.ansi.org/.

[7] *International Organization for Standardization*. URL: https://www.iso.org.

[8] «Cosa sono i microservizi?» In: *Amazon AWS* (). URL: https://aws.amazon.com/it/microservices/.

[9] Martin Fowler. «Microservices: a definition of this new architectural term». In: *www.martinfowler.com* (2014). URL: https://martinfowler.com/articles/microservices.html.

[10] Mayukh Nair. «How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play». In: *Refraction Tech Everything* (2017). URL: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b.

[11] Roy Fielding. «Architectural Styles and the Design of Network-based Software Architectures». Tesi di laurea mag. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[12] W3C. *Web Service Activity*. Rapp. tecn. World Wide Web Consortium, 2002. URL: https://www.w3.org/2002/ws/.

[13] Elliot J. Chikofsky e James H. Cross II. «Reverse Engineering and Design Recovery: A Taxonomy.» In: *IEEE Software* 7.1 (1990), pp. 13–17. URL: http://dblp.uni-trier.de/db/journals/software/software7.html#ChikofskyC90.

[14] Winston W. Royce. «Managing the Development of Large Software Systems». In: (1970).

[15] W3C. *Same Origin Policy*. URL: https://www.w3.org/Security/wiki/Same_Origina_Policy.