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 Booking Healthcare Services


UNIVERSITÀ
DI PARMA

**Relatori**
Prof. Michele Amoretti

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

**Tesi di Laurea di**
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 progetto di Tesi descritto in questo elaborato ha come obiettivo la *reingegnerizzazione di un software utilizzato per la prenotazione di servizi sanitari: Zerocoda.* 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 Zerocoda è attualmente online e operativo, ma non ottimizzato per il suo funzionamento.

Con un'azione di reverse engineering sono stati analizzati i punti critici del sistema, studiandone il traffico e l'effettivo comportamento. Non si è agito 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 questi vengono salvati. Il fine ultimo di quest'opera di reingegnerizzazione è stata la costruzione di un *nuovo layer di Rest APIs* dietro un reverse proxy. Il sistema di partenza 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 può essere causa di un attacco informatico o malfunzionamenti, per questo si è optato per un'architettura di microservizi. Questo nuovo strato è stato affiancato al sistema esistente, fino a quando un successivo sviluppo ne permetterà una completa migrazione. La parte di backend, precedentemente scritta in PHP, è stata riscritta in Java utilizzando la JDK 15 e il framework Spring. Con la reingegnerizzazione il sistema ottiene nuove funzionalità richieste dalle strutture affiliate e migliora quelle già implementate, il tutto *rimanendo coerente con il suo funzionamento*: il nuovo strato inserito tra frontend e backend deve infatti 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 realizzato costituisce il primo passo sulla strada che porterà alla fusione delle due.

# Indice

# Elenco delle figure

# Introduzione

# 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.

## 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*.

### 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 (Figura 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.

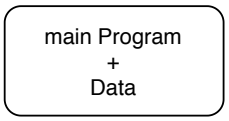



Figura 1. Programmazione non strutturata

### Programmazione procedurale (Figura 2)

Il concetto alla 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.

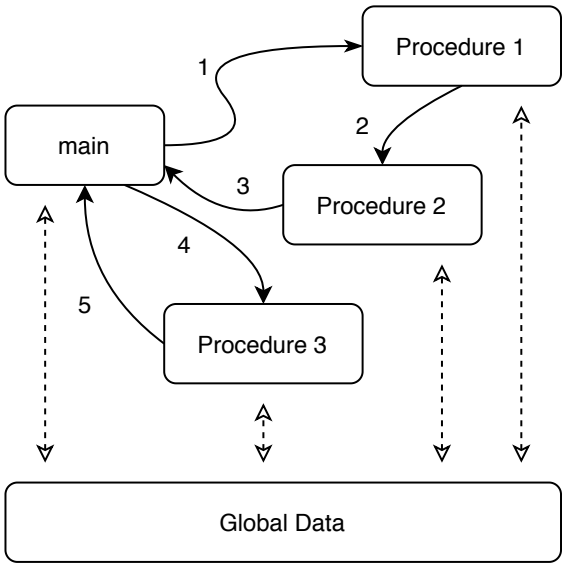



Figura 2. Programmazione procedurale

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 (Figura 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.

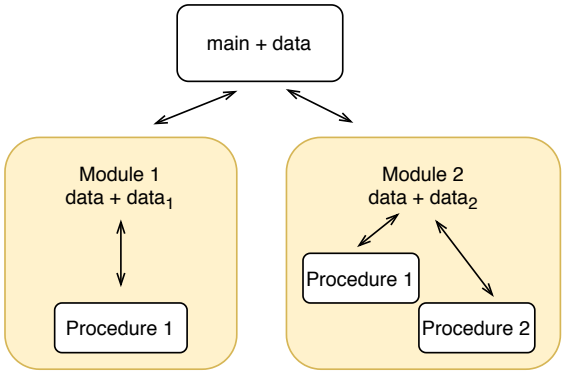



Figura 3. Programmazione modulare

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.

## 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.

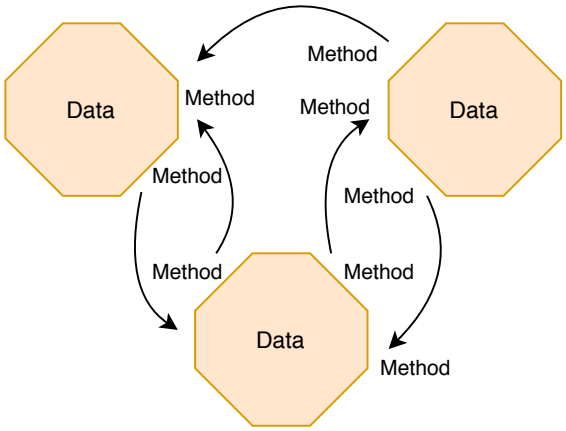



Figura 4. Colloquio tra oggetti

Nella Figura 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.

## 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 5:

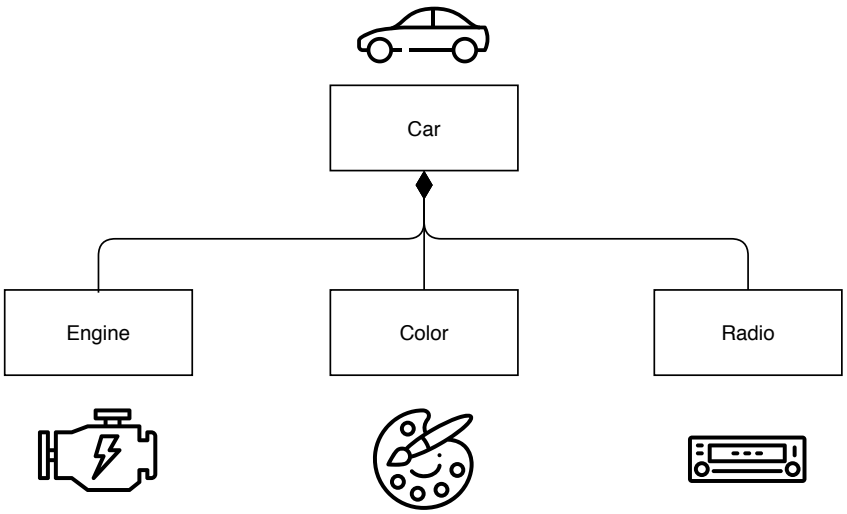



Figura 5. Esempio di un Sistema Orientato agli Oggetti

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.

# 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.

## 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 (Figura 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 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

## 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**.

## 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 7. Esempio di documento di un database non relazionale

## 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.

## 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 (Figura 8), che è l'unico ad accedere fisicamente alle informazioni.

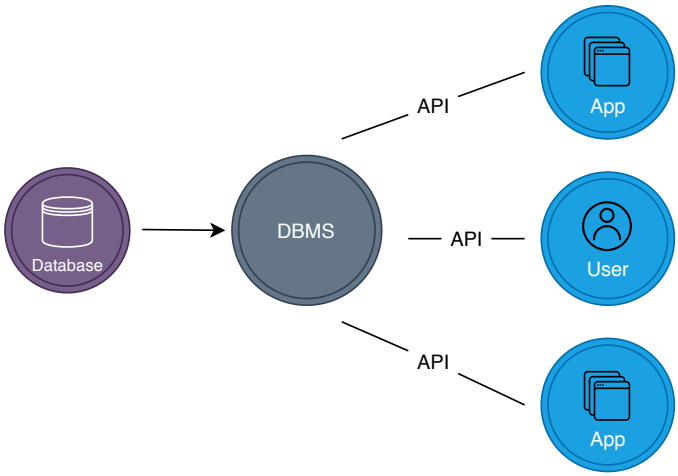



Figura 8. Database management system

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 9. Esempio di Query SQL

# Architettura di Microservizi

## 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.

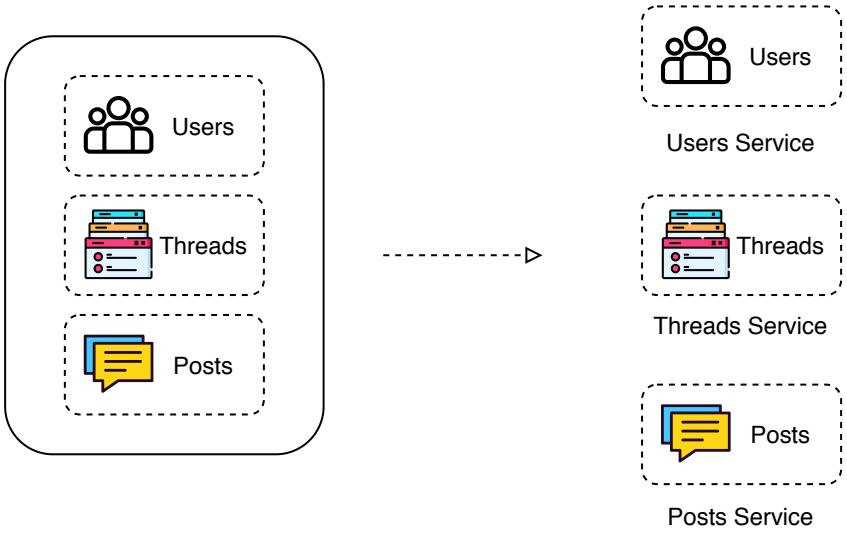



Figura 10. Architettura Monolitica e Microservizi

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.»*

## 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.

## Le differenze con SOA

Per studiare le differenze tra le due architettura facciamo riferimento alla Figura 11:

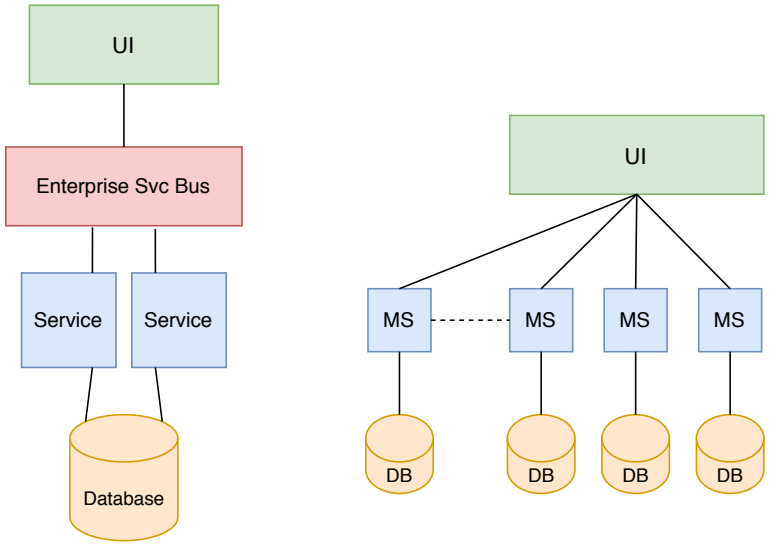



Figura 11. Architettura SOA e Microservizi

**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.

## 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. Gli aggiornamenti sono così rilasciati più velocemente e in modo più sicuro.

### 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.

# Servizi e API

## 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 (Figura 12). 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.

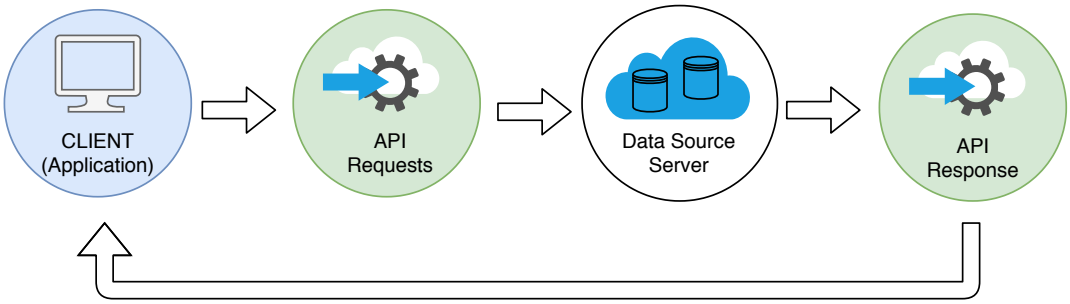



Figura 12. Funzionamento delle API

## 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.

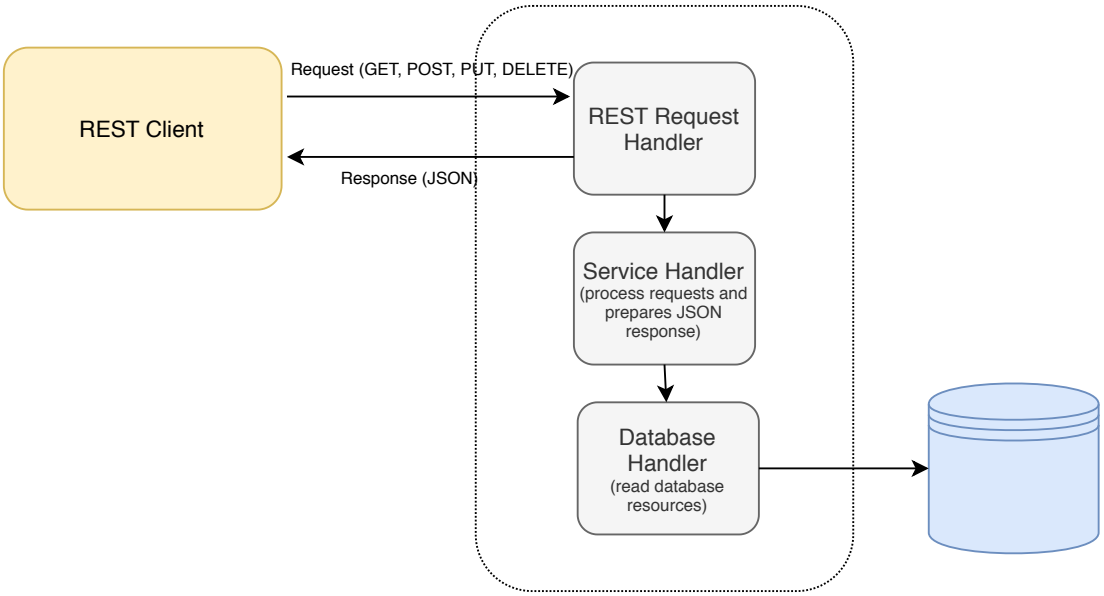



Figura 13. Funzionamento delle API REST

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

Figura 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* $\rightarrow$ *POST*
- *READ* $\rightarrow$ *GET*
- *UPDATE* $\rightarrow$ *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 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.

## 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 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.

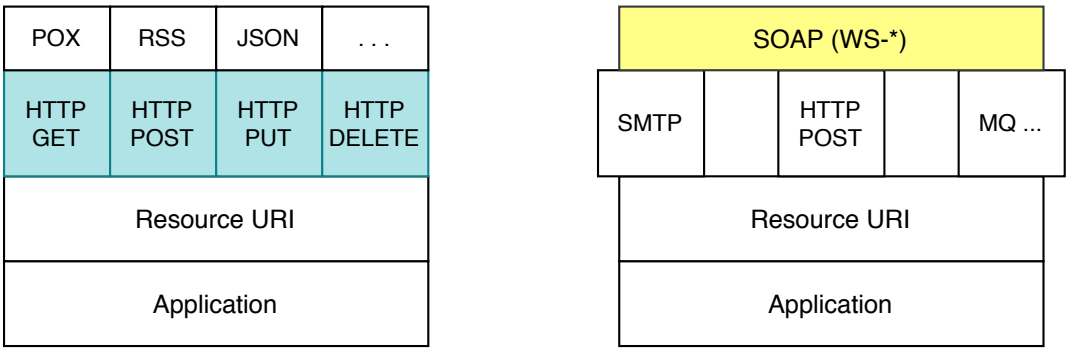



Figura 17. REST vs. SOAP

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 (Figura 17).

# Architettura Funzionale del Sistema

In questo capitolo è illustrato il funzionamento del sistema prima e dopo l'azione di refactoring, presentando le tecnologie che utilizza e accennando alle 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.

## Funzionamento dell'Applicazione

### Scenario

Lo scopo di quest'applicazione è quello di *velocizzare l'accesso ai servizi sanitari*. 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", che è univoco per quel determinato servizio.

**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 si presenti presso la struttura nel momento esatto in cui viene erogato il servizio, evitando coda e assembramenti. In questo modo anche la logistica dietro a un servizio può essere ottimizzata, ad esempio preparando i documenti necessari alla visita.

**La struttura** deve avere informazione di questo biglietto virtuale, così come l'utente. Questo avviene attraverso un *software* installato in loco che prende il nome di **MRU**, dal nome di Mauro e Ruggero, i due ideatori del sistema software.

**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 queste sono presenti le scarica. In questo modo anche gli operatori hanno informazione delle prenotazioni.

## Use Case Diagram

In Figura 18 è presentato il diagramma dei casi d'uso di un utente generico di Zerocoda. I diversi colori utilizzati per le operazioni che l'utente può fare rappresentano la *divisione in servizi* scelta per la fase di implementazione delle REST API.

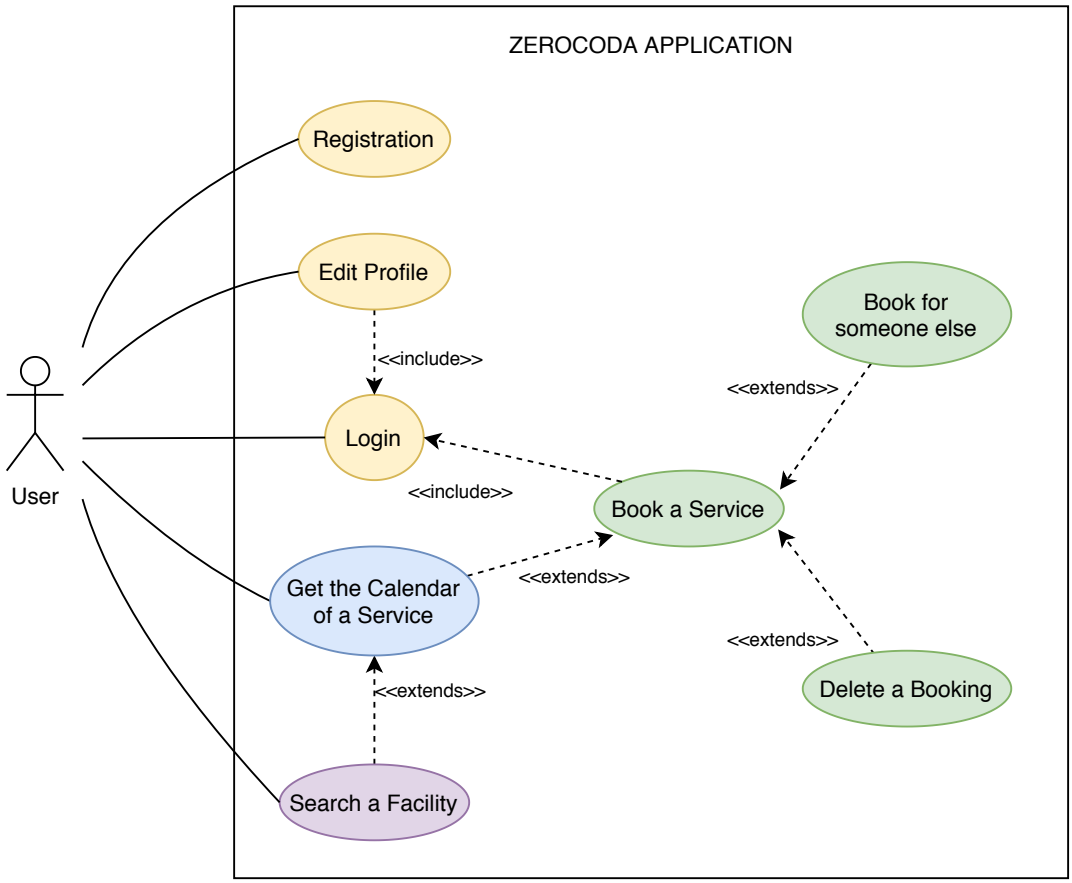



Figura 18. Zerocoda Use Case Diagram

## 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 file 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.

# Analisi dell'Applicazione

Gli attuali 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

## Multitenancy

La particolarità del sistema è quella di essere **multitenant** (Figura 19). 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.

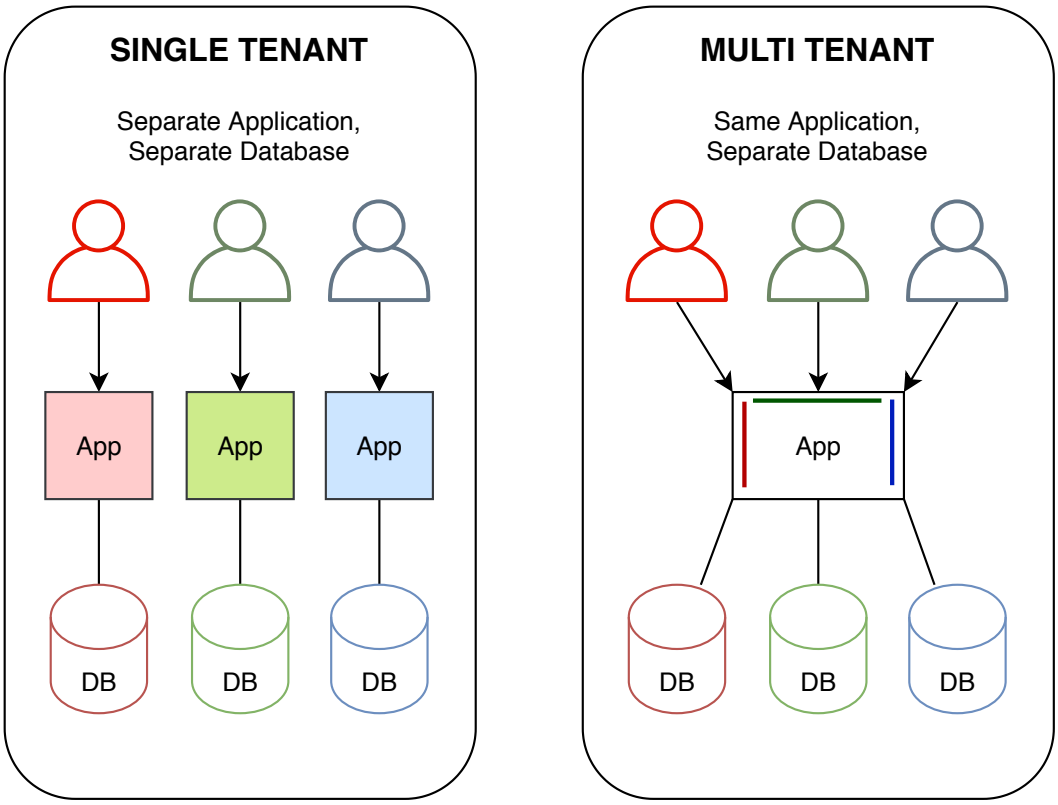



Figura 19. Single Tenant vs. Multi Tenant

## 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.

## 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=mariarossi2020%40studenti.unipr.it&
                    password=Password123!
callback=jQuery22404905003978295608_1605812447764&
                    _=1605812447767
```

Figura 20. Esempio di richiesta GET in HTTP

Prendendo in analisi la richiesta di login in Figura 20, 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 21 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.

### 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. [13]

**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:

```
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 21. Struttura delle chiamate Api

- 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 fa a caso nostro per quanto riguarda le chiamate delle API di Zerocoda, le quali si trovano su un dominio differente rispetto a quello della pagina web. Siccome durante il primo sviluppo dell'applicazione non si aveva pensato ad una soluzione simile, durante la nuova fase di

analisi si è tenuto conto di questo aspetto. Nell'implementazione si lavorerà dunque alla stesura di API su un differente dominio, garantendone l'accesso e dichiarando affidabili le origini necessarie. Alla nascita di Zerocoda si è elaborato un modo per aggirare questo problema, piuttosto che eliminarlo alla radice.

### JSONP

JSONP è l'acronimo di *JSON with Padding* ed è la tecnica utilizzata per ovviare alla limitazione imposta dalla Same Origin Policy. Il suo funzionamento di base è semplice: permette a un browser di accedere a risorse remote mediante codice JavaScript, indipendentemente dall'host di origine. La tecnica permette di invocare una *funzione di callback* automatizzata (come spesso viene fatto per le richieste AJAX basate su XmlHttpRequest) al ricevimento di dati. Il JSON è definito all'interno di questa funzione. Al caricamento del file JavaScript, l'interprete esegue la funzione di callback, che è così in grado di restituire il JSON al client.

```
jQuery22407129987995091923_1605810727465({
    "message":"user logged",
    "status":1,
    "token":"MM5KEZ2S1TKXBNRG8YDI7GP2OQJM8AEOCT49LA4H3IV5WZCFLV",
    "user":{
        "name":"Maria",
        "surname":"Rossi",
        "cf":"RSSMRA80A41G337L",
        "email":"maria.rossi2020@studenti.unipr.it",
        "password_expiration":null}
    }
);
```

Figura 22. JSONP Callback - Login

Nella Figura 22 viene presentata la callback utilizzata per restituire all'utente i propri dati dopo un'autenticazione andata a buon fine. La chiamata AJAX (invocata da JQuery) viene così mascherata con il caricamento di un file JavaScript. La stringa di numeri presente sul parametro callback viene inizializzata con un valore corrispondente al nome randomico di una funzione appena creata da uno script server-side. Come si osserva dalla precedente Figura 20 il server ha conoscenza della funzione da andare a creare, in quanto ottiene questa informazione dai parametri del metodo GET utilizzato. Il codice JavaScript inserito nella pagina HTML utilizza poi questa funzione per passare i parametri che normalmente vengono restituiti con un file JSON. Si tratta sostanzialmente di un'invocazione di funzione.

# Architettura Precedente

In Figura 23 viene rappresenta l'architettura del sistema *precedente* al refactoring. Di seguito vengono analizzate le tecnologie che essa utilizza.

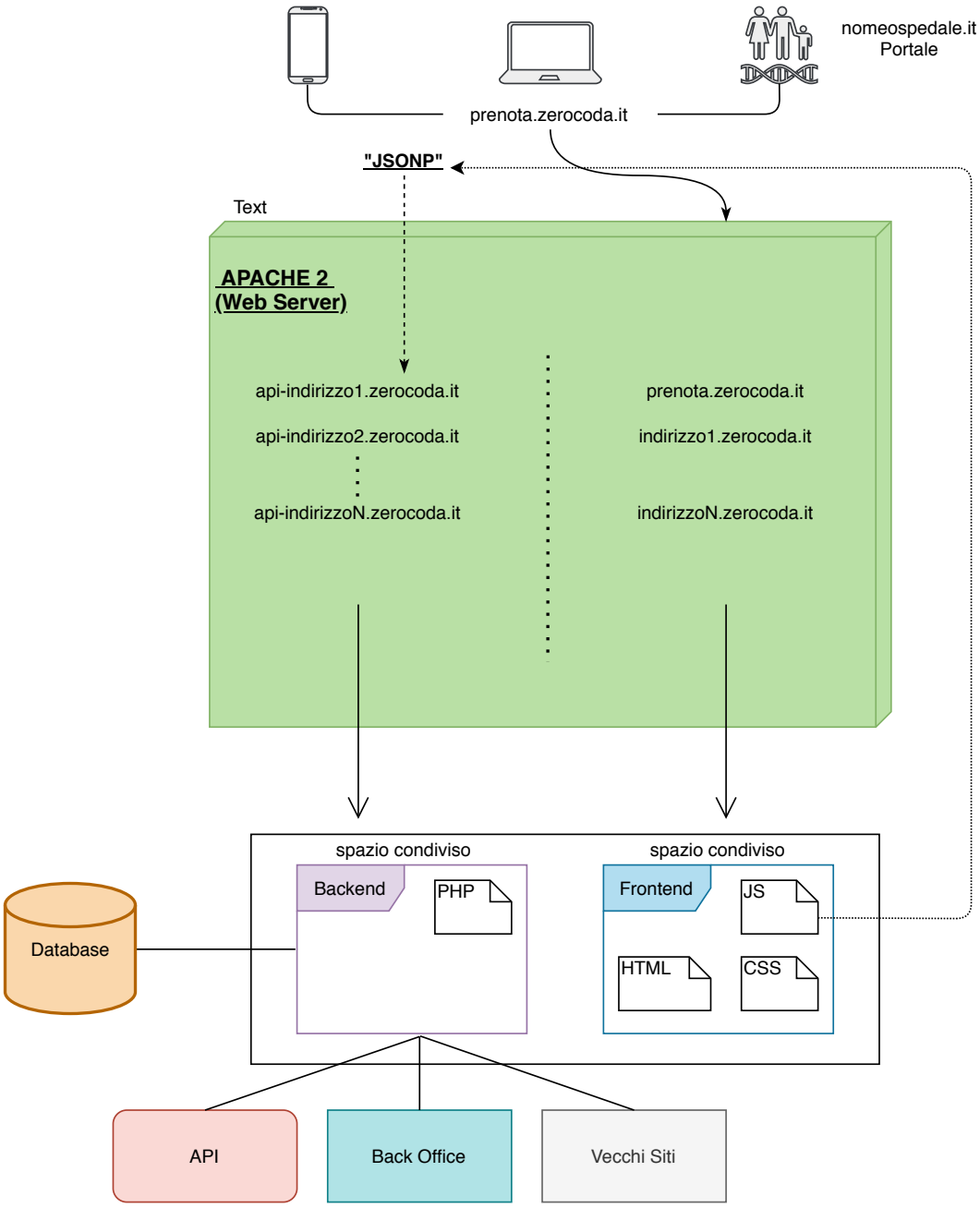



Figura 23. Architettura del Sistema di Partenza

## 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 principalmente per aggiungere quelli che si definiscono 'slot', ma può essere anche utilizzata per creare e gestire utenti. Uno *slot* è 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 (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 modificati per rispondere ad uno stile 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 acceda 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.

## 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, com'è possibile osservare dalla Figura 24 condividono la stessa porta, ma si può anche assegnare a ciascun dominio una porta specifica.

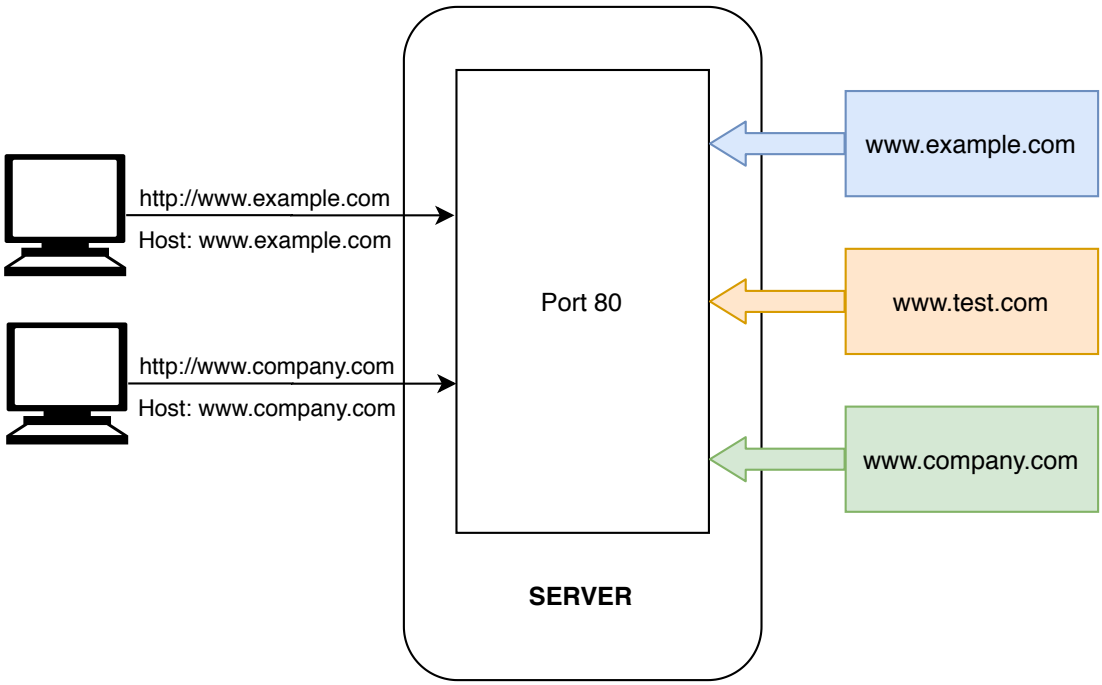



Figura 24. Funzionamento del Virtual Hosting

## Database MySQL

Il dialogo con il database avviene attraverso il codice PHP del backend. È compito del server Apache fornirne l'accesso per una serie di indirizzi (le 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 si limitano ad avere 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.

## 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 configurazione per il software, per lo più utilizzate nel backoffice in fase di amministrazione. Queste tabelle contengono ad esempio i giorni festivi all'interno di un anno (in cui non è

quindi possibile prenotare determinati servizi), o i giorni della settimana per evitare prenotazioni errate nei weekend dove alcune strutture possono essere chiuse. Tra queste sono presenti anche tabelle per la configurazione dei campi utente in fase di registrazione e altre contenti i messaggi di errore da restituire. Questi tipi di tabelle vengono utilizzate quindi come metodo di controllo nelle query. 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.

### Autenticazione

Per autenticarsi l'utente fornisce la propria email e la propria password, che, al momento del primo accesso, contribuiscono alla generazione di un token casuale da parte del sistema. Per gli accessi successivi questo stesso token viene utilizzato per identificare l'utente. In principio Zerocoda era sprovvista di un sistema di autenticazione mediante token randomici. Questa funzionalità è stata introdotta poco prima del refactoring, tuttavia i token di autenticazione vengono salvati sul database e interrogati dallo stesso backend piuttosto che da un server esterno. Nella progettazione della nuova architettura si è pensato a questo problema, gestendolo attraverso un apposito microservizio.

### Verifica delle API Keys

Non è presente nessuna struttura intermediaria tra client e server che si occupi di fare i controlli necessari per la verifica delle API Keys, pertanto queste operazioni si traducono in delle semplici query alla base di dati. L'API Key deve essere un *campo obbligatorio* nella richiesta di un'operazione. Questa permette di identificare l'ente/i corrispondenti, mostrandone i relativi dati sulla pagina. Ad esempio, un utente che accede a Zerocoda deve essere in grado di vedere i soli enti e servizi corrispondenti ad un'API Key, mentre accedendo ad un portale di un sito con una propria configurazione deve vedere i soli dati relativi a prenotazioni e servizi di quella struttura. Anche il controllo dell'Api Key, come per l'autenticazione, si è deciso di delegarlo a un nodo intermedio.

### Ridondanza dei Dati

Come le precedenti, anche la ridondanza dei dati è un limite che ha dato diversi problemi durante la manutenzione del software. Il database presenta due tabelle con lo stesso scopo: *users* e *bookers*. Inizialmente le due erano state pensate con compiti diversi. La prima avrebbe registrato i dati degli *amministratori di sistema*, coloro che hanno necessità di accedere al backoffice, mentre la seconda avrebbe contenuto i dati degli *utenti finali* di Zerocoda. La continua manutenzione degli ultimi anni ha portato il software ad essere sviluppato sempre da *persone diverse*. Questo, unito all'ambiguo nome dato a tabelle, ha

portato i dati ad essere incosistenti da quelli pensati nel disegno orginale. Al momento, i dati di un utente in fase di registrazione vengono quindi salvati su due tabelle distinte.

### Vincoli di Integrità

I vincoli di integrità permettono alle tabelle di disporre di *regole rigide* riguardanti i dati che è possibile inserire al loro interno. Il database non le utilizza in tutte le tabelle. Il modo in cui i dati vengono inseriti o eliminati dal database deve essere sempre lo stesso e coerente negli anni. Non essendo così, anche i metodi di interrogazione ne risentono. Un sistema di questo tipo dovrebbe avere poche query, semplici e comprensibili, tuttavia la disuguaglianza di alcuni valori all'interno del database non lo rende possibile. Alcune prenotazioni, eliminate da utenti o amministratori, presentano valori *null*, mentre altre hanno il valore '0' che le identifica come 'non prenotate'. Quando è necessario eliminare un dato presente su più tabelle, spesso occorre fare più di una query, controllandone il risultato. Questi problemi si sarebbero potuti evitare in fase di progettazione.

## Reingegnerizzazione

**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. [14] 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.

### 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 apportate e si è verificato il fenomeno dell' '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.

# 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.

## 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" [15] 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 vengono utilizzati dei 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. (Figura 25)

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

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.

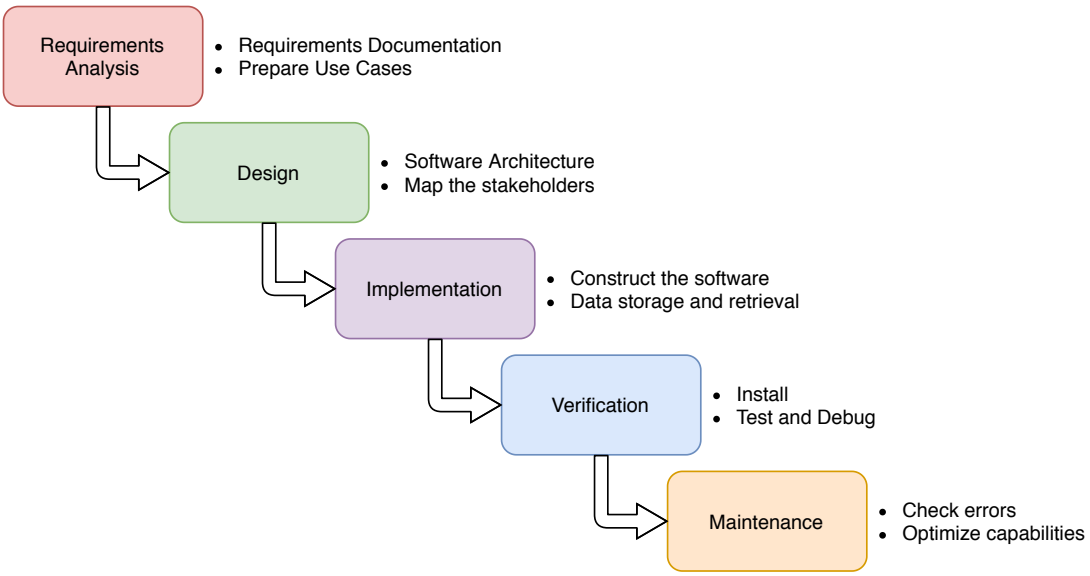
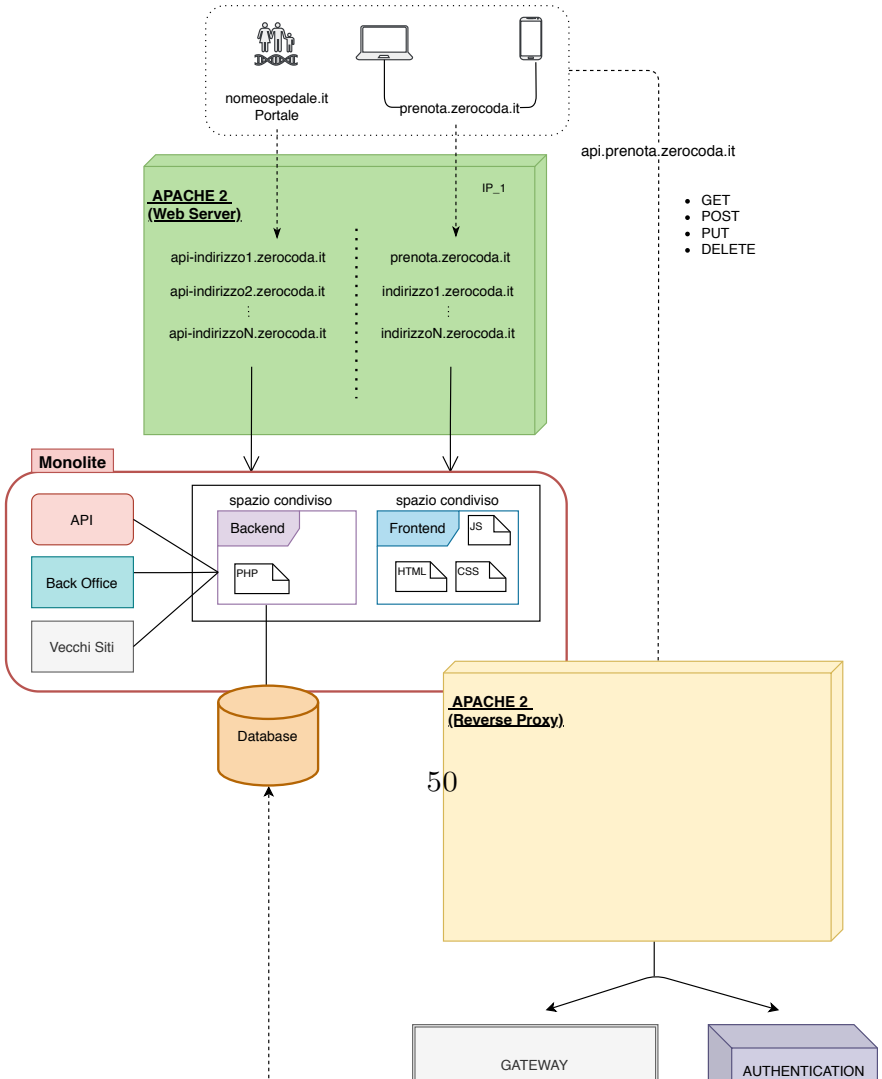



Figura 25. Fasi del Modello a Cascata

# Nuova Architettura



## Monolite

Come si vede in Figura 26 parte della nuova architettura comprende un componente rimasto invariato dall'architettura precedente: il Monolite. Questo infatti, per una prima fase del processo evolutivo, servirà ancora le vecchie API al backoffice. Il monolite verrà meno dall'architettura solo quando comincerà la fase di refactoring inerente al database. Con essa, verrà rifatto il backoffice e le API implementate saranno riadattate con le nuove query. A questo punto anche il backend verrà rimosso, permettendo all'applicazione di utilizzare il Web Server solo per ottenere il frontend.

## Scenario

Quando l'utente si connette al sito *www.prenota.zerocoda.it* viene fatta la risoluzione del DNS e il reindirizzamento sul *vecchio server*. Questo rimane identico al precedente e comprende sempre frontend, backend e database. Durante il caricamento della pagina all'utente vengono sempre restituiti i componenti statici e il JavaScript, aggiornato per utilizzare le nuove API. Quando si deve fare un'operazione si utilizza ora un nuovo indirizzo *api.prenota.zerocoda.it*. L'indirizzo consente le invocazioni di metodi REST, appoggiandosi a un server Apache con un indirizzo IP diverso dal precedente. Quest'ultimo funziona da Reverse Proxy, limitandosi a inoltrare le richieste ad una serie di microservizi. Ciascun microservizio interroga la stessa base di dati, che rimane invariata. Prima anche un ente che avesse richiesto una configurazione con un portale Zerocoda sul proprio sito sarebbe dovuto passare per il Web Server principale. Con l'introduzione delle nuove API sarà possibile utilizzare l'indirizzo fornito, che consente di passare direttamente per il Reverse Proxy.

## Microservizi

La vera innovazione nell'architettura del sistema è data dall'introduzione dei Microservizi. Si è pensato alle operazioni che è possibile svolgere come a 3 macrocategorie di servizi:

- **booking service**: il servizio riguardante le operazioni di prenotazione svolte dall'utente. In particolare questo servizio comprende le operazioni di ricerca delle diverse strutture e dei relativi calendari di disponibilità. Pertanto si è pensato a un ulteriore suddivisione: *Calendar Service* e *Facility Service*.

- **user service**: servizio per l'autenticazione dell'utente e per tutte quelle operazioni relative alla modifica del suo profilo, tra cui la gestione del cambio password.

- **communication service**: il servizio che permette all'utente di essere notificato riguardo allo stato delle sue prenotazioni.

L'implementazione riguarda principalmente la parte relativa al booking service. La parte di comunicazione, insieme a quella di autenticazione, saranno oggetto dello sviluppo in una fase più avanzata del refactoring.

### Gateway

Il Gateway in questa architettura permette di intermediare i microservizi e di esporli in maniera strutturata verso l'esterno. Il suo compito principale è quello di *verifica delle API keys.* Mediante l'utilizzo di queste chiavi il sistema riconosce quali sono i dati da mostrare all'utente, pertanto è importante che questi dati arrivino al server esatti, passando da un primo nodo che ne effettua la verifica. In questo modo il server non viene inutilmente chiamato per la risoluzione di richieste che senza un'apy key valida non è in grado di soddisfare. Il Gateway interessa solo i microservizi di prenotazione e comunicazione, mentre non riguarda quello di autenticazione, che si limita a gestire la sessione dell'utente.

## Authentication Server

Il server di autenticazione ha il compito di rilasciare all'utente il proprio token, che lo identifica univocamente. Questo gli permette di svolgere le operazioni per cui è richiesta un'autenticazione, come quelle di prenotazione di un servizio e la sua relativa cancellazione.

### JSON Web Token

Lo standard utilizzato per l'autenticazione dal server è il *JSON Web Token (JWT).* Si tratta di una tecnologia recente, che consiste nella preparazione di un token con un *payload* che racchiude al suo interno tutte le informazioni relative alla sessione dell'utente. Oltre al payload, nel token viene inserita la firma del server, formata dall'informazione del payload criptata con codifica hash 256, che in questo modo ne autentica le informazioni. A questo punto il token è pronto per l'utilizzo e viene consegnato all'utente, che lo utilizzerà per tutte le successive chiamate. Il client è libero di leggerne il contenuto, ma non può modificarlo perchè in questo modo ne comprometterebbe la firma, rendendolo irriconoscibile per il server. Il server infatti, una volta ricevuto il token, si assicurà che questo sia stato firmato e autenticato con la sua chiave privata, e solo in seguito ne estrapola il payload. Il vero vantaggio nell'utilizzo di questa tecnologia sta nel consistente guadagno in termini di scalabilità per l'applicazione. Poichè il token contieen il payload con tutte le informazioni necessarie all'autenticazione, il server può evitare di passare ogni volta dal datbase per verificare a quale utente questo appartenga.

# Architettura Ideale

In Figura 27 viene presentata l'architettura ideale del sistema progettato. Questa soluzione consiste nel disporre ogni microservizio di un suo proprio database privato, di modo che ciascuno non vada a toccare lo spazio degli altri. Così facendo le operazioni di prenotazione, ad esempio, risultano svincolate da quelle di login e logout. Ogni chiamata a un determinato servizio viene così gestita attraverso l'accesso a uno *spazio non comune di dati*. Questa soluzione porta a un *grande guadagno in termini di scalabilità*, nonchè un immagazzinamento dei dati in maniera più distribuita.

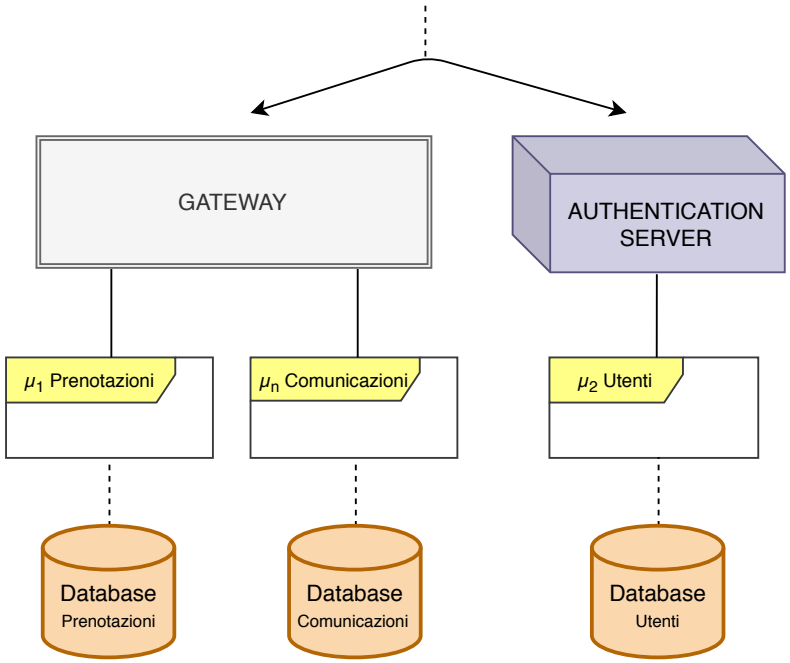



Figura 27. Architettura Ideale del Sistema

Questa condizione risulta necessaria in un contesto di evoluzione del sistema. Oggi, tuttavia, essendo questa prima fase del refactoring non mirata alla ristrutturazione del database, questo tipo di architettura non è attuabile. La sua implementazione risulterà possibile solo in fase di completamento del refactoring.

# Implementazione

In questo capitolo viene descritta l'implementazione del nuovo sistema, partendo dai design pattern a cui si è fatto riferimento per lo sviluppo. In seguito, si utilizza la documentazione per definire le API sviluppate e la loro suddivisione in servizi. Si procede con la presentazione dei framework utilizzati per la creazione dei componenti, presentando le relative parti di codice. Infine si parla della struttura del progetto e dei moduli che lo compongono, analizzando attraverso la simulazione di una chiamata come questi comunicano tra loro.

## Design Pattern

### Inversion of Control

L'*Inversion of Control (IoC)* è il design pattern su cui si basano la maggior parte dei framework moderni, pertanto il nuovo sistema non rappresenta un'eccezione. Normalmente utilizzando una libreria è il nostro codice a richiamare gli elementi definiti all'interno di essa. Con l'introduzione di questo principio è il nostro codice ad essere richiamato dai componenti del framework, da qui il concetto di *'inversione del controllo'*.

#### Façade Pattern

Questo design pattern contribuisce alla semplificazione delle interazioni con il framework o un set complesso di classi. Quando il codice lavora con un vasto set di oggetti appartenenti a librerie o framework sofisticati, normalmente è necessario inizializzare tali oggetti, tenere traccia delle loro dipendenze e chiamare i relativi metodi nel giusto ordine. In questo modo però il nostro sistema risulta strettamente legato all'implementazione di tali framework rendendolo difficile da mantenere. Il pattern risponde a questo problema. Una Facade è una classe che gestisce al posto nostro la logica appena descritta.

Si utilizza lo schema del sito *Refactoring Guru* [16] per semplificare i concetti alla base del suo funzionamento. Prendendo come riferimento la Figura 28 presentiamo gli elementi che compongono il pattern strutturale:

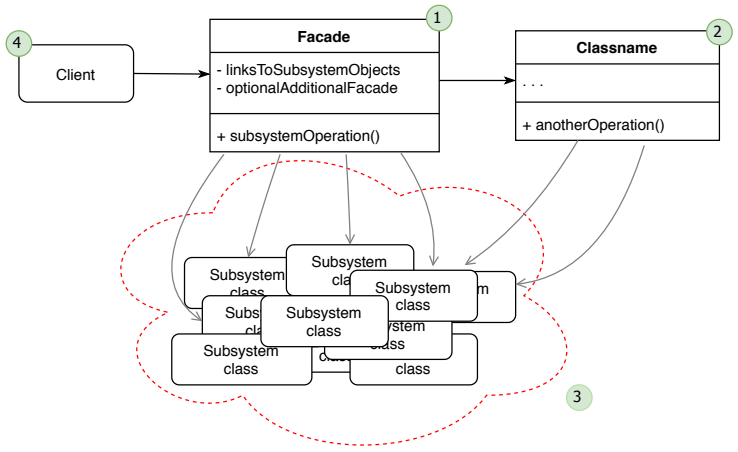



Figura 28. Componenti del Façade Pattern

1. **Facade**: oggetto che dà accesso al set ristretto di funzionalità del Complex Subsystem
2. **Additional Facade**: Facade aggiuntiva nel caso in cui si voglia tenere quella principale il più pulita e semplice possibile
3. **Complex Subsytem**: libreria o framework complesso che vogliamo semplificare
4. **Client**: Il client fa uso del Complex Subsystem attraverso la Facade

# Documentazione

La documentazione è stato il vero primo passo nel percorso di implementazione. Una decisione inusuale per lo sviluppo di un software, ma comprensibile per un caso di refactoring come questo, privo di una documentazione iniziale.

## Swagger

Swagger è un *tool* composto da un set di software open source per progettare, creare e documentare *RESTful APIs* attraverso l'*OpenAPI Specification*, un formato di descrizione apposito per le REST APIs. In particolare aiuta a descrivere:

1. *endpoint* presenti e *operazioni CRUD* che è possibile fare su di essi
2. *parametri* di input e output per ciascuna operazione
3. metodi di *autenticazione*
4. informazioni relative al software, come contatti, licenza e termini di utilizzo

Per descrivere il funzionamento del software si è utilizzato *Swagger UI*, un tool che grazie all'uso delle *annotazioni nel codice* e all'integrazione con il framework Spring, permette di creare una pagina web dinamica che presenta tutte le informazioni relative alle API. Ciò che viene mostrato sulla pagina durante l'esecuzione corrisponde alla specifica delle API. Questo documento può essere anche scritto in YAML o JSON: il formato è facile da imparare e comprensibile sia dall'uomo che dalla macchina.

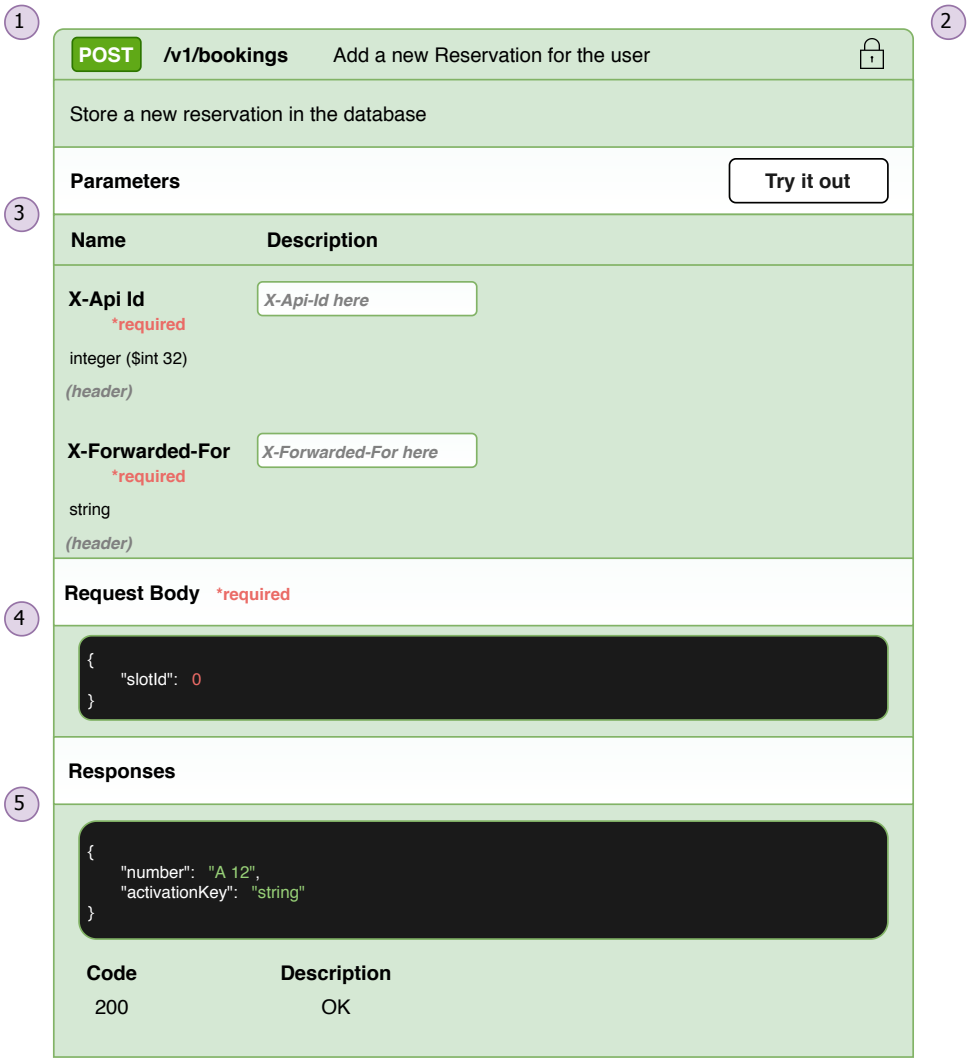



Figura 29. Esempio di Specifica OpenAPI

In Figura 29 viene presentato un esempio di specifica OpenAPI realizzata con Swagger. Il documento descrive l'API di prenotazione di un servizio. In riferimento alla numerazione in figura, elenchiamo i campi che compongono la descrizione della nuova API.

1. Metodo HTTP eseguito sull'url */v1/bookings* dove *'v1'* indica la versione delle attuali API. In questo campo e nel successivo è presente una breve descrizione dell'operazione.

2. Il lucchetto in alto a destra, quando presente, indica che per quella chiamata è richiesta un'autorizzazione. In questo caso l'utente per poter prenotare un servizio deve aver effettuato l'accesso al sistema.

3. In questo campo vengono descritti i parametri e gli header della chiamata. Per ogni parametro viene descritto il suo tipo, e viene aggiunta un informazione che avvisa nel caso questo sia obbligatorio. Di default in tutte le chiamate sono necessari due header: *X-Api-Id* e *X-Forwarded-For*. Il primo passa gli Id API della struttura, così che il sistema sappia sempre quali dati deve mostrare all'utente, mentre il secondo identifica il client che effettua la chiamata al server.

4. Questa sezione descrive il body del messaggio, se presente. In questo caso la chiamata API viene fatta utilizzando il metodo POST, pertanto è previsto un oggetto JSON nel body della richiesta. L'utente per effettuare una prenotazione invia un oggetto contenente l'ID dello slot che vuole prenotare.

5. Nel campo viene rappresentato l'oggetto JSON restituito nel caso la richiesta vada a buon fine. È possibile specificare anche altri codici di risposte che è possibile ricevere.

## Nuove API

Sono state individuate un totale di 6 API associabili al *booking server*, tutte originariamente sviluppate come metodi *GET*. Parte di queste API, oltre alle operazioni sul database, comprendono un sistema di notifica utente. Per queste API si è deciso di separare le due operazioni, delegando il sistema di notifica al *Communication Server*.

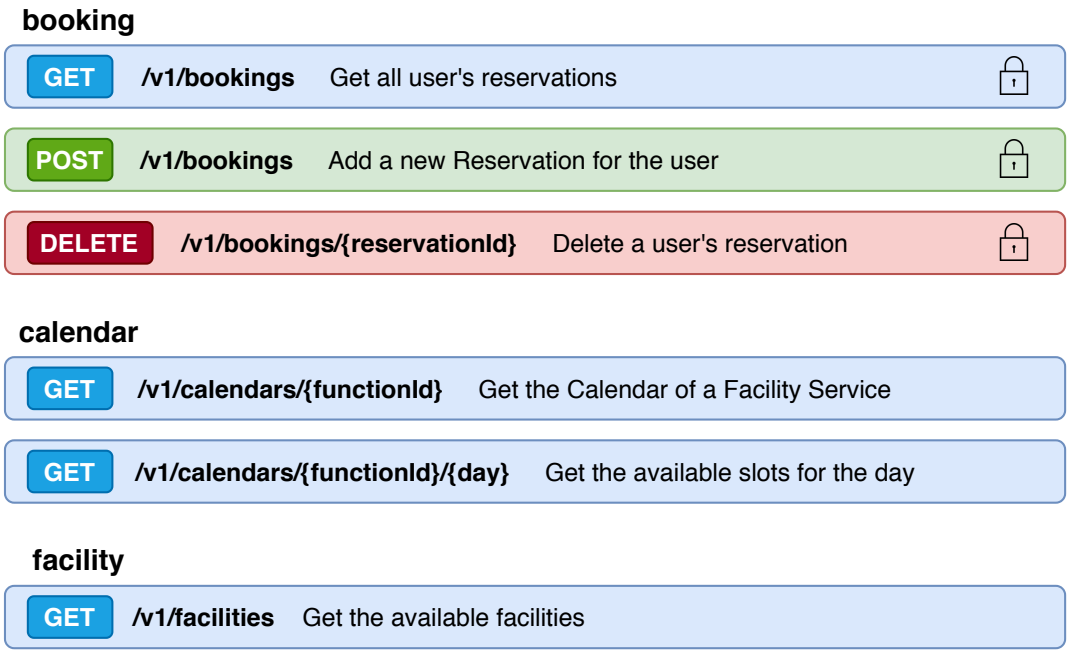



Figura 30. Nuove API REST

In riferimento alla Figura 30, vengono ora elencate le operazioni svolte dalle nuove API REST in ordine di servizio: *booking service*, *calendar service*, *facility service*.

- `GET v1/bookings` recupera le prenotazioni di un utente. Questa operazione richiede un utente autenticato al sistema. Viene chiamata quando l'utente effettua il login o il numero delle sue prenotazioni cambia, come ad esempio dopo una prenotazione andata a buon fine o dopo una sua cancellazione.

- `POST v1/bookings` registra una nuova prenotazione nel database. Questa richiesta è inviata al booking server nel momento in cui l'utente prenota un servizio. Si tratta di una richiesta fatta con il metodo POST, pertanto è previsto l'invio di un oggetto nel *body della richiesta*. Nell'*oggetto Reservation* mandato nella richiesta viene specificato lo *slot* che l'utente vuole prenotare.

- `DELETE v1/bookings/{reservationId}` il metodo delete viene utilizzato nel momento in cui un utente sceglie di eliminare una prenotazione. Nella richiesta viene specificata una *variabile di percorso*, la quale sta a indicare l'ID della prenotazione che l'utente vuole cancellare.

- `GET v1/calendars/{functionId}` restituisce alla pagina i dati relativi al calendario di prenotazione di un servizio (chiamato per l'appunto *function di una facility*). Il

parametro in questo caso rappresenta l'ID del servizio di una struttura. Il calendario viene restituito in termini di data: giorno, mese, anno.

- `GET v1/calendars/{functionId}/{day}` restituisce gli slot disponibili per un determinato giorno appartente al calendario. Può essere considerato come un'estensione del percorso precedente. Ciascuno slot corrisponde a una *fascia oraria* che l'utente può prenotare. L'ID dello slot verrà poi incapsulato dentro un oggetto nel momento in cui l'utente procede con la sua prenotazione, e inserito nel body della relativa richiesta in POST. Gli slot vengono mostrati in termini di orario: ora e minuti.

# Spring Framework

> «Un framework open source nato con l'intento di gestire la complessità nello sviluppo di applicazioni enterprise.»
>
> Team di Sviluppo di Spring

Spring è un framework "leggero" e grazie alla sua architettura estremamente modulare é possibile utilizzarlo nella sua interezza o solo in parte, in quanto non sconvolge l'architettura esistente. Il framework mette a disposizione *una serie completa di strumenti* atti a gestire l'intera complessità di un progetto software. In particolare, fornisce un approccio semplificato alla maggior parte dei problemi ricorrenti nello sviluppo software: accesso al database, gestione delle dipendenze e testing.

## Dependency Injection

La *Dependency Injection (DI)* è una delle diverse implementazioni che la *Inversion of Control* può avere. Spring implementa la *IoC* tramite Dependency Injection. La *DI* prevede che tutti gli oggetti all'interno della nostra applicazione accettino le dipendenze, ovvero gli oggetti di cui hanno bisogno, tramite costruttore o metodi setter. Non sono quindi gli stessi oggetti a creare le proprie dipendenze, ma queste vengono 'iniettate' dall'esterno.

## Annotazioni

La dichiarazione dei componenti (da ora chiamati *bean*), come in generale la configurazione di un'applicazione Spring, può avvenire in tre modi differenti: tramite *file XML*, *classi di configurazione*, oppure *annotazioni Java*. Di seguito si fa riferimento alle principali annotazioni utilizzate durante l'implementazione nel codice Java:

- `@Controller` gestisce le interazioni tra utenti e logica di business. Riceve comandi ed espone le informazioni in modo da essere comprensibili per l'utente o utilizzabile da altri sistemi. Nel codice è stata utilizzata l'annotazione *@RestController* (un alias di questa annotazione) per fare riferimento alle classi di controllo dei servizi *BookingController, CalendarController e FacilityController*.

- `@Service` gestisce la logica di business. Ha la funzione di elaborare i dati e di fornirli ai Controller per essere esposti verso il client. Allo stesso tempo, le informazioni da salvare vengono inviate allo strato di accesso ai dati. Questa annotazione è stata utilizzata nelle classi *BookingService, CalendarService e FacilityService*, dove venivano implementati i metodi chiamati dalle REST API.

- `@Autowired` può essere applicata a diversi elementi della classe: un campo, un metodo o un costruttore. Lo scopo dell'annotazione è quello di indicare a Spring quali sono le dipendenze richieste da un determinato oggetto. Queste vengono ricercate tra le istanze presenti nell'IoC container e, se presenti, vengono 'iniettate' mediante il costruttore o il metodo annotato. Un esempio di implementazione di questa annotazione la si ha nelle classi *Service*, dove per comunicare con il database nel costruttore si istanzia la relativa classe *Data Access Object*, che invece presenta l'annotazione *Service*.

# MyBatis

MyBatis è un framework di persistenza che effettua il mapping (ovvero definisce la corrispondenza) tra i metodi Java e le query SQL. Per farlo, mappa le tabelle presenti sul proprio database con delle classi Java, servendosi di un file di configurazione in linguaggio XML. Le query SQL ovviamente rimangono comunque a carico del programmatore. Il framework si limita a semplificare l'interfacciarsi del codice con il database. Le query vengono inserite in un *file di mapping in formato XML*, nel quale oltre al *comando da eseguire* è necessario specificare le associazioni, ovvero le tabelle che interessano l'operazione sul database.

## Configurazione

```
<generatorConfiguration>
        <jdbcConnection
            driverClass="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/database_name"
            userId="database_id"
            password="password">
        </jdbcConnection>

        <!-- Path to Mapper XML files -->
        <sqlMapGenerator
            targetPackage="it.mapsgroup.zerocoda.booking.persistence.mapper"
            targetProject="./booking-persistence/src/main/resources">
            <property name="enableSubPackages" value="true"/>
        </sqlMapGenerator>

        <!-- Path to Mapper Interfaces -->
        <javaClientGenerator
            type="XMLMAPPER"
            targetPackage="it.mapsgroup.zerocoda.booking.persistence.mapper"
            targetProject="./booking-persistence/src/main/java">
        </javaClientGenerator>

        <!-- Services Table -->
        <table tableName="services">
            <property name="useActualColumnNames" value="false"/>
            <generatedKey column="id" sqlStatement="SELECT LAST_INSERT_ID()"
            identity="true"/>
        </table>
    </context>
</generatorConfiguration>
```

Figura 31. File di Configurazione di MyBatis

In Figura 31 viene mostrato il file XML contenente la configurazione di MyBatis. Nel file individuiamo diversi tag:

- <jdbcConnection> comprende le informazioni di connessione al database. Nel codice mostrato le informazioni sensibili sono state anonimizzate.
- `<table>` specifica una tabella di cui mi MyBatis andrà a effettuare il mapping.
- `<sqlMapGenerator>` specifica il percorso in cui vengono salvati i file XML che mappano una specifica tabella e le sue query.

- `<javaClientGenerator>` specifica il percorso in cui vengono salvate le interfacce corrispondenti ai file XML.

Una volta creato il file di configurazione questo viene passato come input al tool di MyBatis, che procede a effettuarne il mapping appena configurato.

## Creazione di una Query

```
<mapper namespace=".../.mapper.ServicesExMapper">
    <resultMap id="BaseResultMap" type=".../.dto.ServicesEx"
      extends=".../.mapper.ServicesMapper.BaseResultMap">
      <association property="enterprises" columnPrefix="E_"
        resultMap=".../.mapper.EnterprisesMapper.BaseResultMap"/>
    </resultMap>

    <select id="selectByFilter" resultMap="BaseResultMap"
      parameterType=".../.dto.ConfigServicesRemoteFilter">

        select
        S.id as id,
        S.name as name,
        S.enterprises_id as enterprises,
        E.name as E_name


        from enterprises E
        INNER JOIN services S ON E.id = S.enterprise_id
        <where>
            <if test="andServiceIdEqualsTo != null">
                and CSR.id = #{andConfigServicesRemoteIdEqualsTo}
            </if>
        </where>
    </select>
</mapper>
```

Figura 32. Query con MyBatis

In Figura 32 viene rappresentata la sruttura del file *ConfigServicesRemoteExMapper.xml*, che eredita dalla classe padre *ConfigServicesRemoteMapper.xml*. Per i file XML contenti le query delle API si è scelto di utilizzare il suffisso 'ex', in modo da non modificare i file generati da MyBatis, ma limitandosi ad *estendere le loro funzionalità*. La query in figura rappresenta una semplice *JOIN* tra *struttura* e *servizio erogato*. Nel file vengono specificate eventuali associazioni presenti nella query tra la tabella padre e altre tabelle (in questo caso

*ConfigMryouEnterprise*, dove sono salvate le strutture). Nel tag `<select>` vengono specificati *l'attributo id*, che mappa la query, e l'attributo *parameterType*. Quest'ultimo attributo serve per specificare la classe che istanzia l'oggetto che gestisce i parametri inseriti nelle clausole *Group By, Order By, Where*.

```
@Mapper

public interface ConfigServicesRemoteExMapper
    ConfigServicesRemote selectByFilter(ConfigServicesRemoteFilter filter);
}
```

Figura 33. Interfaccia ConfigServicesRemoteExMapper

Una volta creato il file XML, è necessario creare il mapper Java. Il Mapper, come si osserva in Figura 33, è un'interfaccia contenente i metodi corrispondenti agli ID delle query create. Il metodo in questo caso restituisce un unico oggetto *ConfigServicesRemote*. E' possibile fare restituire alle query anche una lista o oggetti come date, stringhe o interi. Il tipo di oggetto risultante deve sempre essere specificato nel mapper XML. In questo caso, per interrogare il sistema, il database ha bisogno di un oggetto ConfigServicesRemoteFilter, creato ad-hoc per semplificare la costruzione della query. La classe di quest'oggetto è costituita da attributi con metodi setter per specificarne il valore. In questo modo, cercando un servizio con uno specifico ID, sarà sufficiente scrivere quanto segue in Figura 34:

```
ConfigServicesRemoteFilter serviceFilter;
serviceFilter = new ConfigServicesRemoteFilter();
serviceFilter.setAndConfigServicesRemoteIdEqualsTo(50);
```

Figura 34. ConfigServicesRemoteFilter Object

L'oggetto creato è ora pronto per essere passato al metodo `selectByFilter`. Così facendo, la query creata nel Mapper XML interroga il database specificando un service ID con valore pari a 50. Le query sono state tutte costruite seguendo questa logica. Nonostante il processo possa sembrare macchinoso sotto certi aspetti, l'utilizzo di MyBatis ha semplificato di molto la comunicazione con il database.

## Struttura del Progetto

In questa sezione viene presentato il packaging del progetto. Si presentano i moduli che lo compongono e le loro interazioni, simulando l'operazione di chiamata di una REST API. All'interno del progetto distinguiamo 3 moduli principali, ciascuno con funzionalità diverse:

- **booking-dto**: contiene le classi utilizzate per instanziare gli oggetti utilizzati nei body delle richieste POST del client, o per istanziare gli oggetti restituiti dal server.

- **booking-persistance**: comprende le classi di persistenza, ovvero tutte quelle classi di Mapping e configurazione delle tabelle del database, nonchè i file XML necessari al corretto funzionamento di MyBatis.

- **booking-server**: contiene le vere e proprie API REST. In questo modulo sono contenute le classi Controller delle API, con i relativi servizi dove vengono implementate le chiamate. In riferimento alla Figura 30 individuiamo 3 controller, ai quali corrispondono i 3 servizi: *booking*, *calendar* e *facility*.

## Data Transfer Object - DTO

Il *Data Transfer Object* è un design pattern utilizzato per trasferire dati tra sottosistemi di un'applicazione software. Nel processo di implementazione, il DTO è stato utilizzato per mappare gli oggetti restituiti dalle query del DAO, in oggetti con cui l'utente può interagire. In questo modo il risultato di una query che sul database interessa diverse tabelle viene presentata all'utente come un singolo oggetto che può comprendere.

### Class Diagram

In Figura 35 viene presentato il diagramma delle classi utilizzate per il mapping dei risultati ottenuti dalle query. Nelle sezioni successive si farà riferimento a questi oggetti. I diversi colori utilizzati rappresentano la suddivisione in servizi. Le classi che non sono state colorate istanziano oggetti che non sono direttamente utilizzati nelle richieste o restituiti nelle risposte dalle chiamate REST.

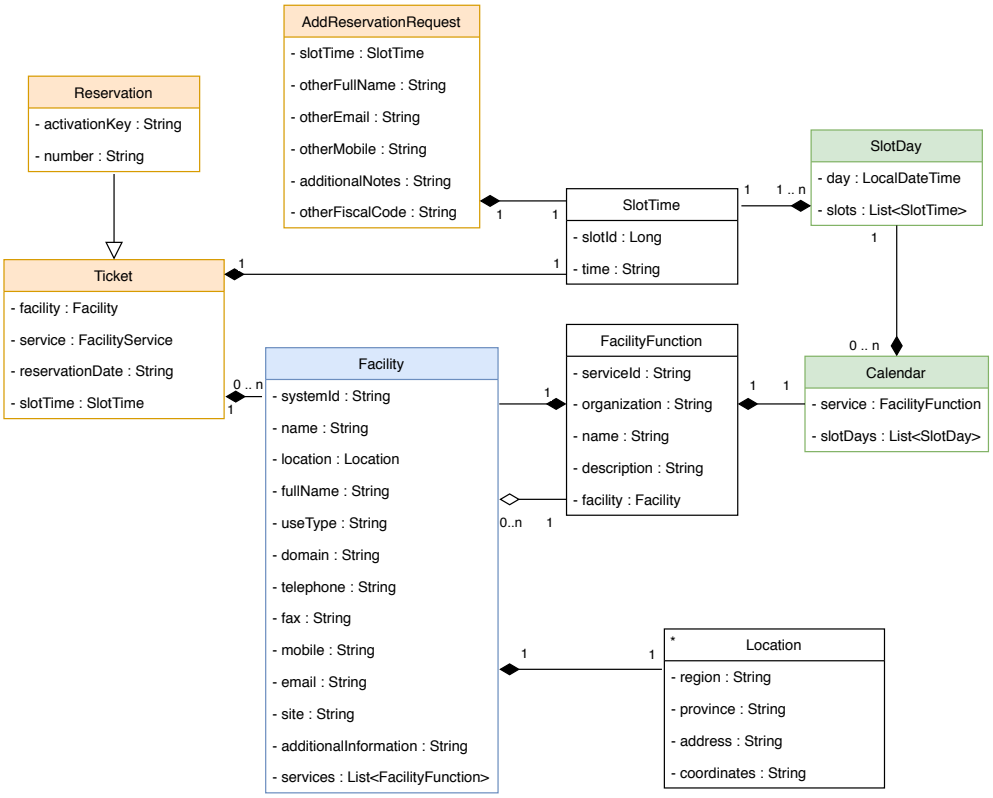



Figura 35. DTO Class Diagram

## Data Access Object - DAO

Il *Data Access Object* rappresenta un pattern architetturale per la gestione della persistenza. Seguendo questo design pattern, è possibile trovare nell'omonimo modulo un set di classi atte a disaccoppire il server dall'accesso al database. Il DAO gestisce l'interazione tra i servizi REST e l'accesso al database mediato dai mapping realizzati con MyBatis, nascondendone i passaggi implementativi.

La classe ConfigMryouEnterpriseDao in Figura 36 viene utilizzata per interfacciarsi con il file di mapping XML creato seguendo la procedura precedentemente esposta. La query chiamata restituisce la lista delle strutture sanitarie, sulla base dell'ID API fornito. L'ID API è un *parametro della query*, che può essere assegnato mediante il relativo metodo setter della classe *ConfigMryouEnterpriseFilter*. La classe utilizza le annotazioni Spring per comunicare con gli altri componenti dell'applicazione.

```
@Service
public class ConfigMryouEnterpriseDao {
private ConfigMryouEnterpriseExMapper mapper;

@Autowired
public ConfigMryouEnterpriseDao(ConfigMryouEnterpriseExMapper mapper) {
    this.mapper = mapper;
}

public List<ConfigMryouEnterpriseEx> findAll() {
    ConfigMryouEnterpriseExample example = new ConfigMryouEnterpriseExample();
    return mapper.selectExByExample(example);
}

public List<ConfigMryouEnterpriseEx> findActive
                        (ConfigMryouEnterpriseFilter filter) {
    return mapper.selectByFilter(filter);
}
```

Figura 36. Classe ConfigMryouEnterpriseDao

## Booking Server

All'interno di questo modulo sono presenti le *classi controller* e le relative *classi service*. Le prime contribuiscono a creare la documentazione Swagger, e quindi le specifiche Open Api (Figura 29), mentre le seconde implementano la chiamata REST.

### Classi Controller

La classe *FacilityController* (Figura 37) presenta la chiamata `GET v1/facilities`: il metodo HTTP che restituisce la lista delle strutture sanitarie. La chiamata è implementata richiamando il metodo dell'omonima *classe service* all'interno della funzione exec.

### Classi Service

A questo punto si conoscono i vari componenti del sistema. I principali componenti che gestiscono la *logica dietro a una chiamata REST* vengono dichiarati come *dipendenze nel costruttore* della classe service. In seguito, queste istanze chiamano le loro funzioni all'interno del metodo che implementa la chiamata. Il service non si limita a preparare l'oggetto della risposta. In questa classe sono analizzati i parametri della richiesta (headers inclusi), e sono lanciate eccezioni se durante l'esecuzione si verifica un problema. Analizziamo il codice della classe *FacilityService*, che comunica con i componenti appena esposti.

```
@RestController
@RequestMapping(value = "/v1/facilities",
                produces = {MediaType.APPLICATION_JSON_VALUE})

public class FacilityController {
    private final FacilityService facilityService;

    @Autowired
    public FacilityController(FacilityService facilityService) {
        this.facilityService = facilityService;
    }

    @SecurityRequirements
    @GetMapping
    @Operation(summary = "Get all available facilities",
            description = "Returns all the available facilities",
            tags = {"facility"})
    @ApiResponses(value = {
            @ApiResponse(responseCode = "404",
                         description = "Facilities not found"),})
    public Result<Facility> facilities(@RequestHeader("X-Api-Id")
                                            Integer apiId,
                                       @RequestHeader ("X-Forwarded-For")
                                            String forwardedFor,
                                       @RequestParam(name = "lang",
                                        required = false,
                                        defaultValue = "")  String lang) {
        return exec("get-facilities",
            () -> facilityService.find(apiId, forwardedFor, lang));
    }
}
```

Figura 37. Classe FacilityController

In Figura 38 viene mostrata l'implementazione della chiamata `GET v1/facilities`. Com'è logico pensare la classe utilizza le annotazioni Spring per legarsi alla classe ConfigMryouEnterpriseDao (illustrata precedentemente) e al DtoMapper, una classe creata appositamente per convertire gli oggetti mappati sui file XML di MyBatis in oggetti definiti nel DTO (Figura 35). Nella classe viene prima creato il filtro come visto (Figura 34) e successivamente chiamata la query definita nel classe DAO (Figura 36). Se il risultato della query ha una struttura particolarmente difficile per essere convertito dal DTO, può essere passato al DaoHelper come in questo caso. Il DaoHelper è una classe che ha il compito di separare gli oggetti risultanti dalle query con una o più condizioni di join. Così facendo è possibile fornire al DtoMapper anche due oggetti distinti, facilitandone la conversione. Il nuovo oggetto restituito dal DtoMapper viene in seguito incapsulato in una *lista Result*,

che contiene due elementi: la lista contenente gli oggetti restituiti dal mapper (in questo caso sono oggetti Facility) e il numero totale degli elementi della lista.

```
@Service
public class FacilityService {
    private static final Logger LOG = getLogger(FacilityService.class);
    private final ConfigMryouEnterpriseDao configMryouEnterpriseDao;
    private final DtoMapper dtoMapper;

    @Autowired
    public FacilityService(ConfigMryouEnterpriseDao configMryouEnterpriseDao,
                           DtoMapper dtoMapper) {
        this.configMryouEnterpriseDao = configMryouEnterpriseDao;
        this.dtoMapper = dtoMapper;
    }

    public Result<Facility> find(Integer apiId, String forwardedFor,
                                                String lang) {
        LOG.debug("Looking for facilities [apiId={}, forwardedFor={}]",
                                            apiId, forwardedFor);
        ConfigMryouEnterpriseFilter filter = new ConfigMryouEnterpriseFilter();

        // filter settings
        filter.setAndIsActive(true);
        filter.setAndApiKey(apiId);
        filter.setAndVisibleInWeeks(1);
        filter.setGroupByClause("CME.id, CSR.id");
        filter.setOrderByClause("CME.id, CSR.id");

        List<ConfigMryouEnterpriseEx> enterprises;
        enterprises = configMryouEnterpriseDao.findActive(filter);

        List<Tuple2<ConfigMryouEnterprise, List<ConfigServicesRemote>>>
                            orderedEnterprises =
                DaoHelper.orderFacilities(enterprises);

        List<Facility> ret = orderedEnterprises.stream()
                .map(t -> dtoMapper.copy(t.first(), t.second(), lang))
                .collect(Collectors.toList());
        return new Result<>(ret, ret.size());

    }
}
```

Figura 38. Classe FacilityService

# Attività di Test

In questo capitolo vengono descritti i testeffettuati per valutare le prestazioni del sistema sviluppato. Per fare questo genere di test è stato utilizzato *JMeter*: un tool per misurare le prestazioni di un sistema mediante dei test di carico. Durante questi test si confrontano le vecchie API con le nuove, descritte nel Capitolo .

## Configurazione di JMeter

*JMeter* è un progetto open source dell'*Apache Foundation*. La vera particolarità di questo software è quella di essere perfettamente configurabile per automatizzare le operazioni di test di carico su macchine remote. Nel nostro caso è stato utilizzato per testare le richieste, mediante le API, al database. Sono stati realizzati due progetti: il primo relativo alle *vecchie API di Zerocoda*, il secondo relativo alle *nuove*, oggetto di questo elaborato. E' stata simulata l'azione concorrenziale di più utenti (thread) che attraverso un dataset di valori pre-inseriti andassero a richiamare le API. Per una corretta interpretazione dei risultati, la configurazione è rimasta la stessa tra i due progetti. È rimasto *invariato il numero di thread*, l'intervallo (in secondi) in cui questi venivano chiamati, e il numero di loop (serie di operazione) eseguito da ogni thread. Le operazioni svolte da ciascun test mirano a simulare l'attività di un utente sul sistema, e in ordine sono:

1. Ricerca delle strutture
2. Ricerca del calendario di una struttura
3. Ricerca degli slot di un calendario
4. Prenotazione di un servizio
5. Cancellazione della prenotazione appena effettuata

I dati inseriti sono stati gestiti attraverso diversi dataset, che a ogni loop fornivano a ciascun thread il parametro da utilizzare nelle richieste. Per evitare errori, in entrambi i progetti si è fatto sì che i thread non prenotassero nello stesso momento lo stesso servizio.

La richiesta, una volta effettuata la prenotazione, ne restituiva l'identificativo, che veniva utilizzato dalla richiesta successiva per cancellare la prenotazione dell'utente.

# Risultati Ottenuti

Di seguito vengono illustrati i grafici realizzati con appositi plugin di JMeter. Di seguito vengono illustrati i grafici realizzati con appositi plugin di JMeter. Si fa riferimento a due tipologie grafico:

- **Response Time Graph**: il tempo di risposta in millisecondi di ogni richiesta (asse delle ordinate) viene calcolato in funzione dell'orario in cui la richiesta è stata effettuata (asse delle ascisse)
- **Aggregate Graph**: per ogni richiesta (asse delle ascisse) viene presentata la media dei tempi di risposta in millisecondi (asse delle ordinate)

## Vecchie API

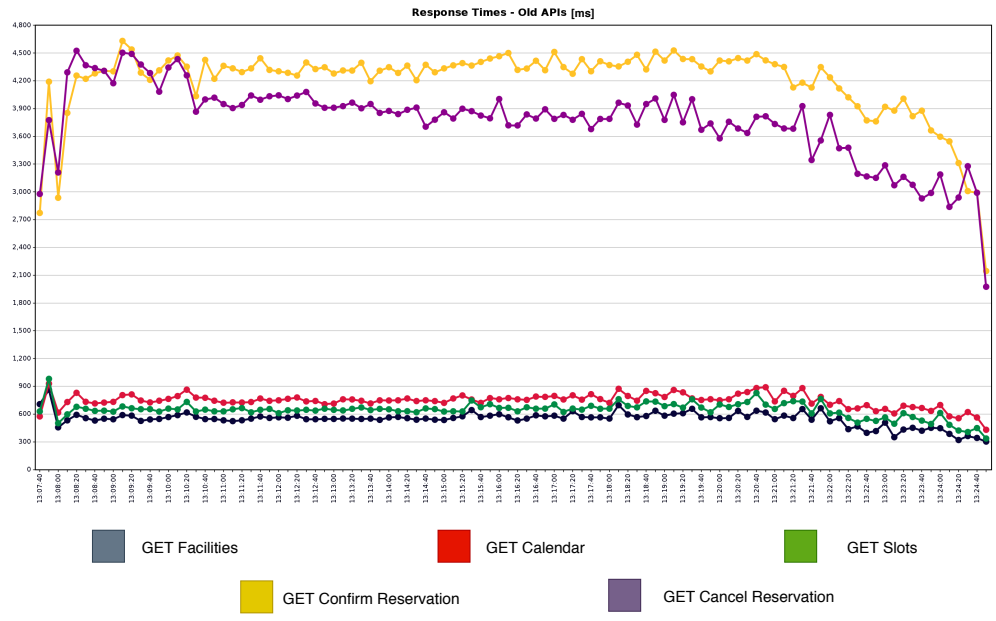



Figura 39. Response Times Graph - Old API

Le API precedenti al refactoring sono messe molto alla prova già con un centinaio di utenti che operano sul sistema (Figura 39). Per le operazioni di lettura sul database vengono impiegati circa *800 millisecondi*, mentre il tempo si alza per le operazioni di scrittura, dove

una richiesta può impiegare fino a quasi *5000 millisecondi* per restituire una risposta. Nella tabellla mostrata in Figura 40 viene mostrata le media (in millisecondi) dei tempi di risposta per ogni richiesta. Anche le richieste più semplici possono impiegare fino a *1000 millisecondi* per l'elaborazione, se il sistema viene messo sotto sforzo.

| Request | # Samples | Average | Min | Max | Throughput |
|---|---|---|---|---|---|
| GET Facilities | 10000 | 547 | 29 | 6625 | 9,65525 |
| GET Calendar | 10000 | 745 | 37 | 6546 | 9,65532 |
| GET Slots | 10000 | 642 | 31 | 4129 | 9,65546 |
| GET Conf. Res. | 10000 | 4205 | 72 | 9978 | 9,65418 |
| GET Can. Res | 10000 | 3773 | 96 | 9432 | 9,65309 |
| TOTAL | 50000 | 1983 | 29 | 9978 | 48,24383 |

Figura 40. Sommario dei Tempi del Test *[ms]* - Vecchie API

Per questo test, così come per il successivo, si è testato uno scenario in cui 100 utenti (threads) eseguono in parallelo le chiamate presentate, per un totale di 100 volte. I *samples* rappresentano il numero di loop in cui è stata eseguita ciascuna richiesta. Si osservino i parametri ottenuti nel *throughput*, e si tengano a mente per un confronto con i corrispettivi valori nelle nuove API.

## Nuove API

Le nuove API hanno dato un esito positivo già nel loro tempo totale di esecuzione. I test sono stati eseguiti separatamente, in modo che l'accesso al database da parte di un backend non andasse a influire sulla richiesta di connessione da parte del backend rivale. Il test svolto con le nuove API ha avuto un *tempo di esecuzione quasi 20 volte minore al precedente.* Un risultato straordinario, ma sostenuto da diverse ragioni. In Figura 41 viene presentato il grafico relativo ai tempi di risposta di ciascuna chiamata. I tempi in millisecondi, rispetto al grafico precedente, risultano parecchio ridotti. Anche le chiamate con i metodi POST o DELETE, che vanno ad effettuare delle operazioni di scrittura sul database, hanno comunque dei tempi minori rispetto a una qualsiasi richiesta GET che nelle vecchie API si limitasse a leggere dei valori. Tutte le richieste nel complesso hanno presentato un miglioramento. Se nel primo grafico a seguito di un picco il sistema stabilizzava le risposte su quel tempo, ora a un picco segue un riabbassamento, e *la gestione delle connessioni appare più controllata.* Il picco è limitato a un istante di tempo, e in seguito ad esso il sistema abbassa nuovamente i propri tempi di risposta.

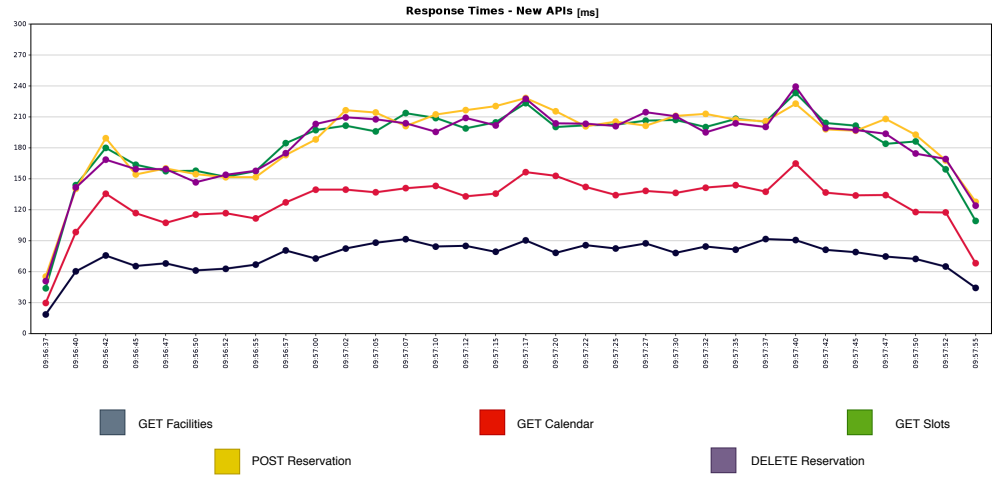



Figura 41. Response Times Graph - New API

| Request | # Samples | Average | Min | Max | Throughput |
|---|---|---|---|---|---|
| GET Facilities | 10000 | 89 | 28 | 108 | 100,43993 |
| GET Calendar | 10000 | 150 | 49 | 187 | 100,44699 |
| GET Slots | 10000 | 223 | 76 | 260 | 100.44295 |
| POST Reser. | 10000 | 249 | 98 | 283 | 100,44598 |
| DELETE Reserv. | 10000 | 232 | 92 | 311 | 100,45203 |
| TOTAL | 50000 | 189 | 28 | 311 | 502,00299 |

Figura 42. Sommario dei Tempi del Test *[ms]* - Vecchie API

In Figura 42 vengono mostrati in una tabella i dati relativi al grafico precedente. Le nuove API REST hanno abbassato di molto la media dei tempi di risposta del sistema, *aumentando di diverse unità il throughput*. Questo valore può essere considerato un *indice di misura delle prestazioni del sistema*. Viene misurato in *bit/s* e rappresenta la quantità di informazioni elaborate in un secondo. La differenza tra i due valori è evidente, ma dietro questi numeri c'è da chiedersi: *come fa il sistema dietro le nuove API a garantire dei tempi di risposta tanto più bassi?*

## Confronto dei Sistemi

Vengono illustrati i grafici in Figura 43 e in Figura 44 rappresentanti la media dei tempi delle due API, a cui si fa riferimento.

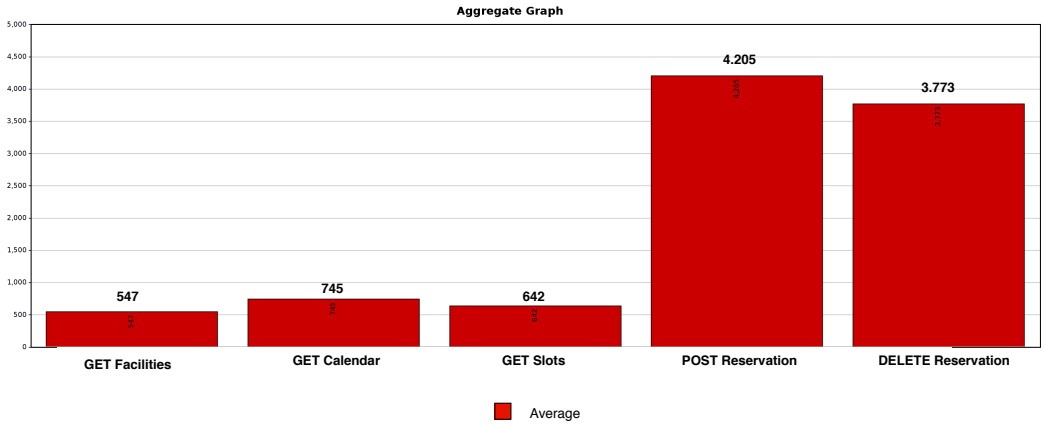



Figura 43. Aggregate Graph - Old API

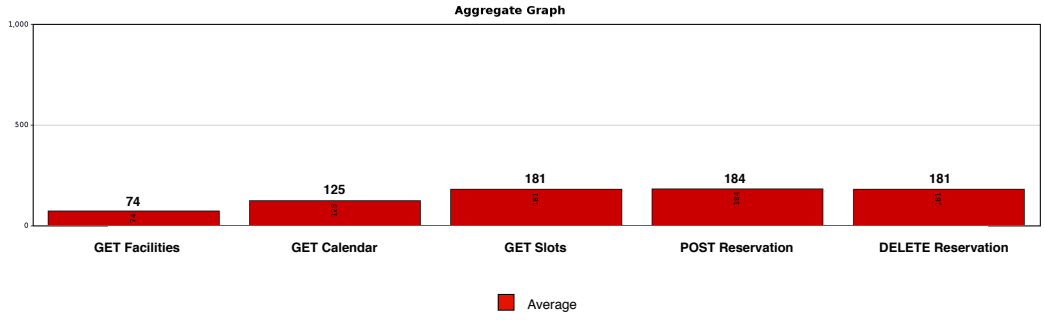



Figura 44. Aggregate Graph - New API

La ragione principale degli ottimi risultati ottenuti con la nuova implementazione è banale: l'utilizzo di Java. Il linguaggio rappresenta la prima differenza che salta all'occhio tra i due backend. In PHP, linguaggio usato nella vecchia versione del backend, per ogni richiesta di accesso al database, che fosse questa una richiesta di lettura, scrittura o modifica dei dati, viene creata un'apposita nuova connessione. La connessione viene poi utilizzata dalla chiamata per svolgere l'operazione desiderata, e in seguito chiusa. In Java questa stessa modalità viene migliorata. Ad ogni richiesta è sempre assegnata una connessione, ma spetta al sistema il compito di decidere *a chi* e *quando* assegnarla. Attraverso una *connection pool* le connessioni al database vengono mantenute per velocizzare gli accessi futuri e l'esecuzione dei comandi. In questo modo, ad ogni richiesta non viene più istanziata una nuova connessione, ma a questa ne viene passata una già esistente. Il sistema all'avvio apre un numero ben definito di connessioni al database, e queste vengono assegnate mano a mano alle richieste che ne richiedono l'accesso. In questo modo il backend dietro le nuove

REST API riesce sempre a garantire una risposta bassa in termini di millisecondi, il suo tempo di esecuzione risulta ridotto rispetto alla vecchia tecnologia, e il valore di throughput ottenuto risulta nettamente migliore rispetto al precedente.

# Conclusioni

Durante lo sviluppo dell'elaborato sono stati affrontati diversi temi. Nello Stato dell'Arte (Capitolo ) sono state presentate le tecnologie utilizzate dai moduli che compongono i due sistemi. Queste sono state confrontate, per studiare quale fosse la soluzione migliore per il nuovo obiettivo. Lo studio, presentato nel Capitolo  si è rivelato necessario per conoscere in particolare quali fossero i *rischi* derivanti dall'adozione di nuove tecnologie, e per capire se convenisse cambiare quelle del sistema precedente nella sua totalità o solo in parte. Sulla base di questi ragionamenti è stata sviluppata la nuova architettura del sistema. Il tutto è stato fatto tenendo conto di una certezza: il sistema implementato sarebbe dovuto coesistere, per le prime fasi del refactoring, con la *relativa parte legacy*. Nell'architettura del sistema implementato, pertanto è rimasto il 'monolite' che adempie a diversi compiti e che rappresenta sostanzialmente il backend di partenza. Questo backend, assieme alle relative API, rimane in appoggio al backoffice, l'interfaccia amministrativa attraverso la quale è possibile aggiungere, per ogni servizio, nuovi appuntamenti prenotabili dall'utente. I processi implementativi dietro la realizzazione della nuova architettura sono stati descritti nel Capitolo . Qui sono stati presentati i design pattern seguiti durante lo sviluppo del software e le tecnologie utilizzate, mostrando il funzionamento di una chiamata e il modo in cui i vari componenti del sistema comunicano tra loro. Grazie ai test effettuati per valutare le prestazioni del sistema nel Capitolo  si è constatato che il sistema è pronto ad esssere trasferito sul *branch di produzione*. Il nuovo software è in grado di gestire un carico di utenze maggiore rispetto al precedente, e con tempi di risposta migliori. Il sogno di poter vedere Zerocoda, un'applicazione da sempre caratterizzata da un'architettura monolitica, lavorare attraverso microservizi è ora diventato realtà. L'applicazione ha così raggiunto quella *scalabilità* tanto perseguita in corso di sviluppo, quanto assente in partenza. La Tesi ha quindi raggiunto il suo scopo, in quanto la reingegnerizzazione del software ha portato all'ottenimento di un sistema più performante.

## Sviluppi Futuri

Il prossimo step consisterà quindi nell'installazione del software. Il layer di API implementato verrà spostato sul branch di produzione, in appoggio al Gateway e all'Authentication Server. Il primo utilizzato per il controllo delle chiavi API, mentre il secondo per la gestione dei dati utente in maniera autonoma, per ciascun servizio, dal database. Il miglioramento ottenuto in termini di prestazioni, tuttavia risulterà davvero palpabile solo quando anche *le ultime parti dipendenti dal monolite si appoggeranno al reverse proxy davanti al layer di API.* Questo segnerà il vero completamento dell'azione di refactoring. Per far sì che ciò accada ci sono ancora due fasi da seguire. Il frontend, rimasto inalterato durante lo sviluppo di questo elaborato, dovrà essere soggetto a modifiche per rispondere alle esigenze grafiche ora dettate dalle nuove API. Il database, d'altra parte, dovrà anch'esso subire dei cambiamenti. Le query delle nuove API sono state riscritte, tuttavia è il sistema di salvataggio dei dati che sul database che necessita un cambiamento. Oltre a subire una completa ristrutturazione, per il database verranno studiati i vincoli di integrità migliori, atti a garantirne il corretto funzionamento nel tempo. Al completamento di questa ultima fase il sistema risulterà pronto all'integrazione finale con Zerocoda Royalty.

## Considerazioni

La pandemia dell'ultimo anno ha contribuito alla messa in luce dei limiti di software come questo. I tagli alla sanità non hanno colpito solo le risorse ospedaliere, ma anche software che come Zerocoda propongono piccoli miglioramenti a degli scenari di vita quotidiana, per le aziende e per le persone. Il mancato impiego di risorse in applicazioni di questo tipo ha fatto sì che le loro tecnologie utilizzate rimanessero indietro rispetto a quelle di altri settori. Solo negli ultimi mesi lo sviluppo di questa applicazioni sono state favorite dalla situazione, che ha permesso loro di avere la giusta importanza nello scenario attuale. Il mancato investimento nello sviluppo di software come questo può portare a una risoluzione dei suoi problemi mediante dei *work around* piuttosto che all'effettivo studio di una soluzione. È evidente che la tecnologia ha sempre più un ruolo chiave nella nostra società, sia per i privati che per le aziende. Investire in questo settore può portare a un'efficienza dei servizi mai vista prima, permettendo alle stesse strutture di essere preparati a casi-limite come quello dell'ultimo anno. Due sono le qualità chiave da perseguire nello sviluppo software: *efficienza* ed *scalabilità*. Il sistema deve sì essere manutenibile nel tempo, ma prima deve essere facilmente aggiornabile all'introduzione di nuove tecnologie, in quanto:

*Il software è come l'entropia. É difficile da afferrare, non pesa nulla, e obbedisce alla seconda legge della termodinamica: aumenta sempre.*

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

[14] 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.

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

[16] *Facade Pattern*. URL: https://refactoring.guru/design-patterns/facade.