**Università degli Studi di Padova**

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica


# Analisi, miglioramento e ampliamento delle funzionalità di un Booking Engine per la prenotazione di crociere

*Tesi di laurea triennale*

*Relatore*

Prof. Luigi De Giovanni

*Laureando*

Michele Tagliabue

Anno Accademico 2017-2018

# Sommario

Il presente documento relaziona il prodotto del lavoro, della durata di circa trecentodieci ore, svolto dal laureando Michele Tagliabue presso l'azienda WebPD s.r.l. durante il periodo di stage. Prima dell'inizio dell'attività sono stati fissati diversi obiettivi da raggiungere.
Per cominciare, si era pianificato di imparare ad interagire, tramite le librerie del framework Codeigniter, con il database relazionale SQL Server. Successivamente, si era prefissato di integrare nel Booking Engine un nuovo fornitore (Royal Caribbean). Nello specifico, si era prefissato di realizzare l'integrazione del flat-file (catalogo) fornito da Royal Caribbean, l'aggiunta delle crociere di Royal Caribbean ai risultati di ricerca, la comunicazione con i Web Service di Royal Caribbean per la sincronizzazione della disponibilità di cabine (e prezzi) e per la prenotazione di queste ultime. Infine, era stato previsto lo sviluppo di un sistema di registro "carichi/scarichi" (magazzino) di tariffe "vuoto per pieno" e di eventuali funzionalità aggiuntive.

Questa tesi si compone di quattro capitoli. Nel primo viene delineato il profilo dell'azienda e le metodologie di lavoro della stessa. Il secondo capitolo presenta (anche a livello tecnologico) il progetto al centro delle attività svolte durante lo stage, che verranno approfondite (suddivise in base agli obiettivi) nel terzo capitolo. Infine, il quarto capitolo presenta una valutazione retrospettiva del tirocinio, sia a livello oggettivo, considerando, ad esempio, il grado di soddisfacimento degli obiettivi, che soggettivo, esponendo, quindi, una mia valutazione personale su quanto svolto.

## Convenzioni tipografiche

Durante la stesura del testo sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati sono definiti nel glossario, situato alla fine del presente documento e ogni occorrenza è evidenziata in blu, come l'esempio seguente: DBMS;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere corsivo.

# Ringraziamenti

*Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Luigi De Giovanni, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.*

*Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.*

*Voglio ringraziare anche la mia ex insegnante delle superiori, professoressa Anna Maria Zottis, per avermi trasmesso la passione per la programmazione. È grazie a lei che ho iniziato a programmare, ed è probabilmente anche grazie a lei che sono riuscito a compiere questo percorso di studi.*

*Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.*

*Padova, Settembre 2018* Michele Tagliabue

# Indice

# Elenco delle figure

# Capitolo 1

# L'azienda

## 1.1 Profilo

Web PD s.a.s. è una software house nata nel 2009, con sede a Padova ma con clienti in tutta Italia (logo in Figura 1.1). La società nasce con lo scopo di gestire portali e-commerce, ma negli anni ha modificato la sua *key activity* in sviluppo di soluzioni software personalizzate. Ad oggi si occupa di consulenza informatica su software gestionali, applicazioni web e mobile (sia Android che iOS).


**Figura 1.1:** Logo WebPD

Il primo prodotto realizzato da Web PD è il gestionale Martina, un software per piattaforma Windows, sviluppato secondo un'architettura client-server modulare, che consente la gestione del ciclo attivo (vendite) e passivo (acquisti) di un'impresa commerciale. Grazie alla sua struttura modulare, è possibile aggiungere a Martina diverse funzionalità, come ad esempio gestione di magazzino, gestione della contabilità e gestione dello scadenziario.

Successivamente l'azienda aggiunge al portfolio di servizi offerti anche la realizzazione, grazie alle piattaforme OpenCart e Prestashop, di siti e-commerce integrati al gestionale Martina.
È in questo contesto che prima nascono gli e-commerce TuttoNauticaWeb.com e MiglioNautico.com, successivamente FarmaZero.com e SubitoStore.com.

Nel 2015 Web PD da s.a.s. si trasforma in s.r.l. perché ha l'intenzione di addentrarsi nel settore turistico; questa strategia si perfeziona con l'acquisizione di un' agenzia di viaggi ad Albignasego (PD).

Nel 2016 il network e franchising di agenzie di viaggio Primarete Viaggi e Vacanze s.r.l. acquisisce il 49% delle quote sociali di Web PD s.r.l.. Il frutto di questa

collaborazione è l'ammodernamento, continuo ed ancora in corso, dei portali Viaggi-regalo.it, CrociereRegalo.it, SimaWorldTravel.it e PercorsiReligiosi.it, di proprietà di Promoter Travel s.r.l., una controllata di Primarete.

## 1.2 Dominio tecnologico

L'azienda offre un ampio ventaglio di prodotti, che coinvolgono diverse piattaforme. È opportuno esaminare le tecnologie adoperate raggruppandole nelle seguenti categorie: *Web*, *Mobile* e *Desktop*.

### 1.2.1 Web

In WebPD, la realizzazione di prodotti basati su piattaforma Web avviene, a livello di *backend*, utilizzando come linguaggio di programmazione PHP (versione 5 o 7). Molto spesso, per i progetti più complessi, viene usato il framework *Codeigniter* (versione 3, il cui funzionamento è illustrato in Figura 1.2), che agevola l'implementazione del design pattern MVC e fornisce numerosi strumenti per facilitare (dunque accelerare) lo sviluppo. Codeigniter, tra le altre cose, infatti, implementa una serie di classi che facilitano l'esecuzione ed il debug delle query, oltre ad un meccanismo che permette di salvarle in cache (aumentando notevolmente le prestazioni del prodotto).

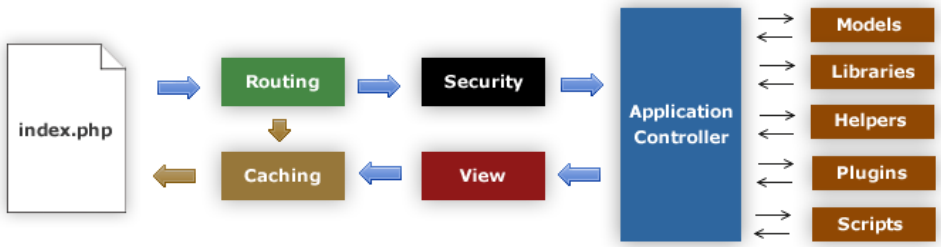


**Figura 1.2:** Flow chart di un'applicazione creata con Codeigniter.
URL: https://bit.ly/2QHOOrz

Per quanto riguarda il *frontend*, invece, viene utilizzato HTML5, CSS3 e Javascript (jQuery), senza usare framework che implementino Single Page Application (come React.js o Angular).

Come server web, infine, vengono usati *Apache2*, in caso di *hosting* su piattaforma Linux, o *IIS*, in caso di *hosting* su piattaforma Windows.

Per la realizzazione di siti web semplici, dove per semplici si intende senza esigenze di svolgimento di operazioni complesse, viene prediletto l'utilizzo di CMS, quali Wordpress (nel caso di blog, vetrine o forum) e OpenChart (nel caso di siti di e-commerce).

#### DBMS

Per lo sviluppo di prodotti software, WebPD si appoggia a due DBMS: *Microsoft SQL Server 2012* e *MySQL Server*. Entrambi questi software sono RDBMS, ma la scelta di utilizzare l'uno o l'altro dipende dalle caratteristiche del progetto, nello specifico:

* **Microsoft SQL Server** viene utilizzato nei progetti che prevedono un'ampia mole di dati da gestire e interrogazioni (query) complesse (come, ad esempio, CrociereRegalo.it), in quanto in tali contesti si è dimostrato mediamente più performante di *MySQL* [2];
* **MySQL Server** viene utilizzato il più possibile in caso il progetto non contenga una grossa mole di dati e/o interrogazioni complesse, in quanto non prevede costi di licenza (a differenza di *SQL Server* che ha costi abbastanza elevati [1]) ed è multipiattaforma (attualmente disponibile per Linux, Windows e MacOS).

### 1.2.2 Mobile

Il *know-how* accumulato da WebPD nella realizzazione di applicazioni web ha portato all'adozione del framework *Cordova* (il cui schema di funzionamento è illustrato in Figura 1.3) per la creazione di applicazioni mobile, il quale offre la possibilità di creare app ibride cross platform utilizzando HTML/CSS e Javascript. Tali applicazioni, grazie ad alcune interfacce messe a disposizione da Cordova, possono accedere alle funzionalità native del dispositivo, come fotocamera, storage o sensoristica varia (accelerometro, giroscopio e GPS, se presenti).

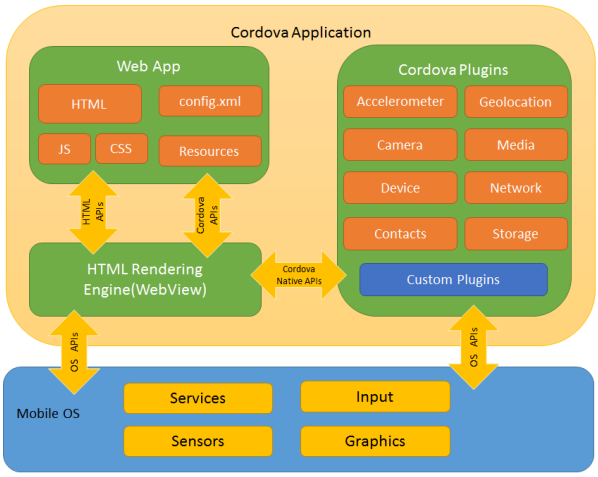


**Figura 1.3:** Schema dell'architettura di un'app basata su Cordova.
URL: https://bit.ly/1TWONeg

### 1.2.3 Desktop

La realizzazione di applicazioni desktop avviene attraverso l'utilizzo di due linguaggi: Java e Delphi.
Delphi viene usato soprattutto quando i programmi necessitano di manipolare grandi basi di dati, in quanto le applicazioni sono compilate in binario, quindi mediamente

più performanti, e il linguaggio possiede numerose librerie che ne facilitano l'accesso. Java, di contro, viene usato quando si ha l'esigenza di creare applicazioni cross-platform che non coinvolgano grandi volumi di dati.

## 1.3 Processi aziendali

### 1.3.1 Ciclo di vita

In WebPD, la *way-of-working* inerente al ciclo di vita del software aderisce al *modello Evolutivo*, le cui caratteristiche sono illustrate in Figura 1.4.
L'adozione di tale modello deriva principalmente dall'esigenza di doversi adattare alla mutevolezza dei requisiti. È un dato di fatto, purtroppo, che molto spesso neanche il cliente stesso ha chiare le proprie esigenze, quindi solo "toccando con mano" l'istanza del suo pensiero esso sa dire se effettivamente è ciò che fa per lui o no.

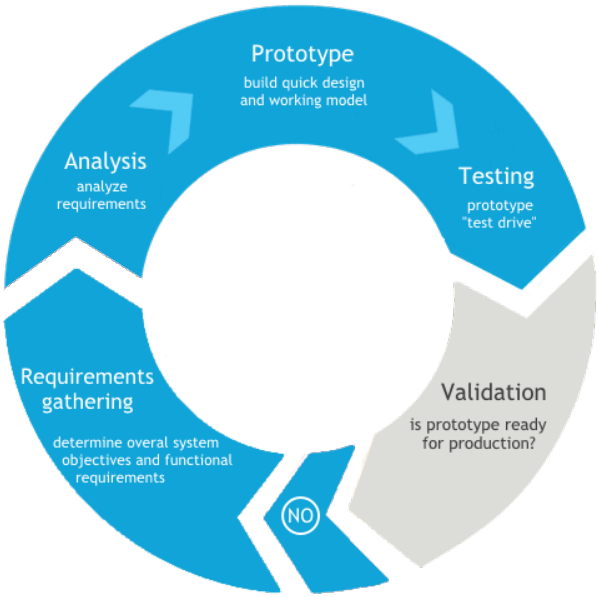


**Figura 1.4:** Schema del modello evolutivo. URL: https://bit.ly/2N49cUI

Questo problema viene affrontato dal modello evolutivo, prevedendo che non venga effettuato solo un unico rilascio, ma che vengano creati prototipi, istanze di un sottoinsieme di requisiti del progetto, sottomessi poi al cliente. L'esito della valutazione dei prototipi può portare ad una rivisitazione dei requisiti inerenti alle funzionalità contenute in essi.
Nello specifico, il modello prevede le seguenti fasi :

1. **Analisi e progettazione**, che viene fatta solo all'inizio del progetto e non si ripete;
2. Per ogni prototipo:
   (a) **Progettazione di dettaglio** dei requisiti non implementati;
   (b) **Sviluppo** di quanto appena progettato nel dettaglio;

(c) **Validazione** del prototipo appena sviluppato e, in base ai feedback del cliente, un'eventuale **revisione dei requisiti**.

3. **Rilascio** della versione finale, corrispondente all'ultimo prototipo accettato dal proponente.

Il modello evolutivo, inoltre, prevede la possibilità di avere più canali di sviluppo (ad esempio stabile, alpha, beta ecc.), utili per poter "sperimentare" nuove funzionalità senza andare ad intaccare la versione stabile del software.
Il problema principale di questo modello è che i tempi di sviluppo possono allungarsi notevolmente. La validazione da parte del cliente, infatti, è un'operazione che può richiedere molto tempo, durante il quale lo sviluppo non può proseguire. Inoltre, in caso di un cliente molto "indeciso", il numero di prototipi può aumentare, rischiando quindi che gli incrementi si trasformino in iterazioni.

### 1.3.2 Miglioramento della qualità dei processi

Nella way-of-working aziendale è presente una forte attenzione alla qualità dei processi. L'organizzazione interna di questi ultimi è incentrata sul principio del miglioramento continuo, grazie all'applicazione del Ciclo di Deming, conosciuto anche con l'acronimo PDCA. Tale acronimo contiene le iniziali delle quattro fasi in cui è possibile suddividere il processo di miglioramento:

1. **Plan**, ovvero pianificare, prima dell'inizio del processo, attraverso una attenta analisi di esso, eventuali problematiche e azioni di miglioramento;
2. **Do**, ovvero eseguire il processo monitorato agendo secondo quanto pianificato nella fase precedente;
3. **Check**, ovvero valutare (in modo retrospettivo) l'esito delle azioni derivanti da quanto pianificato all'inizio;
4. **Act**, ovvero agire standardizzando ciò che è andato a buon fine e colmando le carenze.

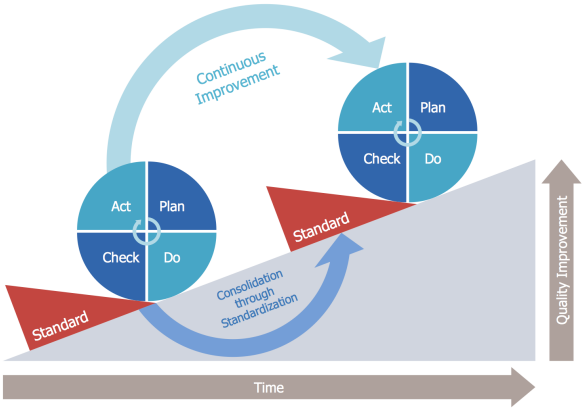


**Figura 1.5:** Rappresentazione grafica del PDCA. URL: https://bit.ly/2LP32TF

A livello grafico, il PDCA è rappresentato attraverso un cerchio in movimento, che dichiara la ciclicità della sua applicazione.

In Figura 1.5 lo standard è rappresentato come un cuneo in quanto agisce da limite inferiore per la qualità. L'obiettivo del PDCA è quindi portare sempre più in alto la qualità dello standard.

### 1.3.3 Strumenti a supporto dei processi

#### Gestione di progetto

Per quanto riguarda gli strumenti di gestione di progetto, WebPD si affida a *Trello*. Trello è uno software di amministrazione di progetto web-based, con numerose applicazioni per Android e iOS. Trello permette la creazione di una bacheca condivisa (come mostrato in Figura 1.6), organizzata in *cards*, all'interno delle quali è possibile inserire dei compiti da svolgere (*task*) e assegnare ciascuno di questi compiti una data di scadenza.

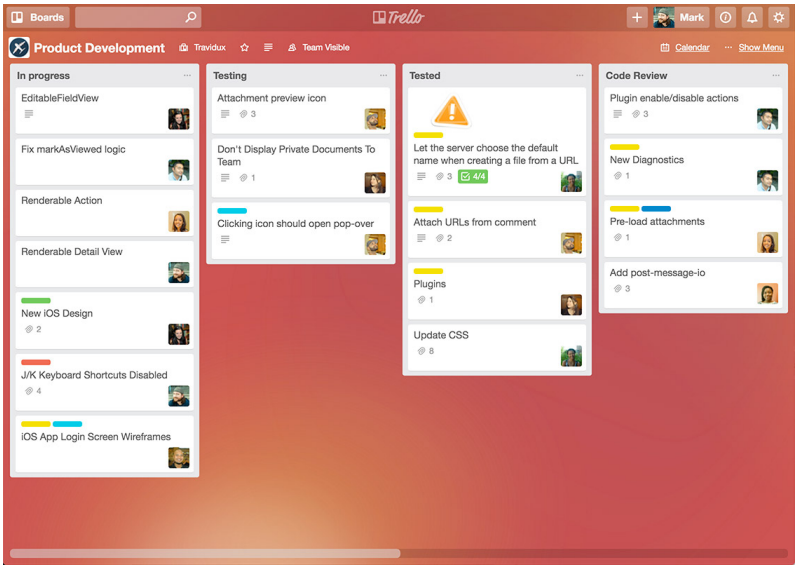


**Figura 1.6:** Screenshot dell'interfaccia web di Trello.

Su ogni compito è possibile pubblicare dei commenti per, ad esempio, porre domande o fare determinate osservazioni e aggiungere checklist, etichette (tag) e allegati.
I *task* possono inoltre essere agilmente spostati tra una *card* e un'altra, funzionalità utile, ad esempio, per tenere quanto più pulita possibile la bacheca, nascondendo i *task* già svolti o raggruppandoli in apposite *cards* aventi funzione di storico.
Una feature molto interessante è la sincronizzazione in tempo reale delle modifiche, che permette a più persone di agire e collaborare sulla bacheca contemporaneamente, senza incorrere nel rischio di visualizzare o modificare dati non aggiornati.
Trello, infine, prevede anche numerosi *Power-up*, ovvero estensioni (a pagamento) che permettono di interagire con numerosi servizi, quali, ad esempio, Google Drive, Onedrive, Google Calendar, GitHub, BitBucket, Slack ecc.

Per avere una più immediata visualizzazione dell'andamento dei *task*, assieme a Trello viene usato *Trellogantt*. Quest'ultimo, come forse già suggerisce il nome, permette

di visualizzare i dati contenuti in una bacheca di Trello sotto forma di diagrammi di Gantt, come mostrato in Figura 1.7. Inoltre permette, con un semplice drag-and-drop, di modificare le date di scadenza dei vari task.

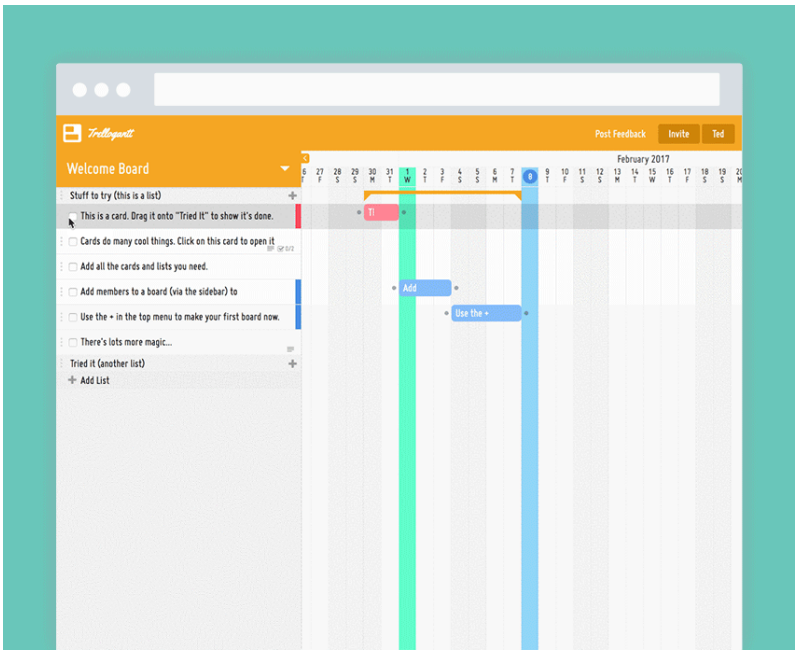


**Figura 1.7:** Interfaccia di Trellogantt. URL: `https://bit.ly/2wysjgc`

### Gestione di versione

WebPD, come strumento di versionamento, utilizza il software *Git* (logo in Figura 1.8), probabilmente il più utilizzato nel suo campo.


**Figura 1.8:** Logo di Git

Le motivazioni che hanno spinto WebPD all'adozione di Git al posto del suo "antagonista" *Subversion* (SVN) sono illustrate in Figura 1.9. In particolare [3]:

* Decentralizzazione del repository: a differenza di Subversion, l'architettura di Git prevede che vi sia una copia dell'intero repository in ciascun client, in modo da poter effettuare *commit* anche in assenza di una connessione verso il server (cosa non possibile con SVN);

* Velocità: le operazioni di commit sono molto spesso più veloci su Git in quanto vengono effettuate sulla copia locale del repository, ed è possibile poi inviarle al repository remoto (tramite *push*) in un secondo momento, anche in blocco.
* Miglior gestione di branch e tag: in SVN, branch e tag sono copie dell'intero progetto (che quindi possono aumentare molto lo spazio richiesto dal repository sul server), mentre in Git sono semplicemente dei riferimenti ad una determinata commit.

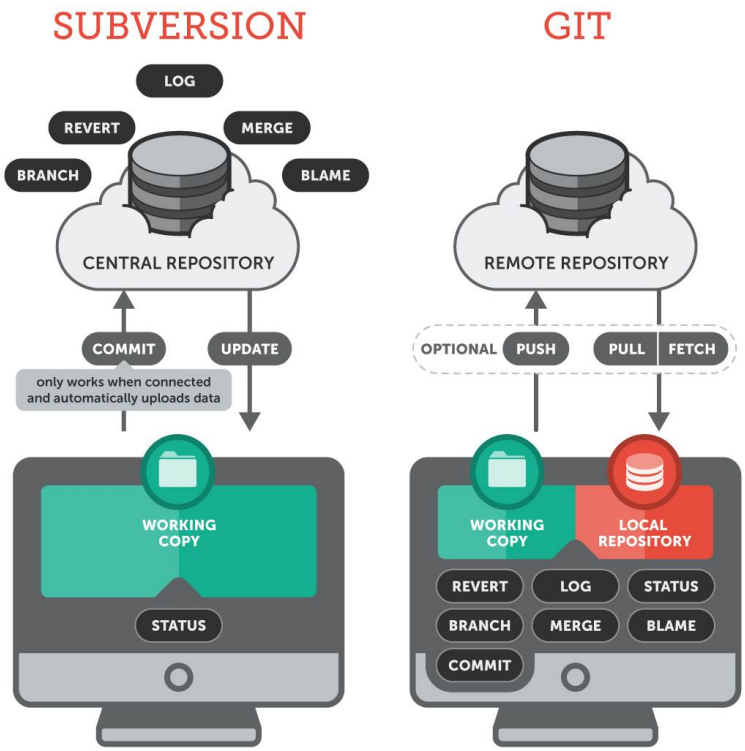


**Figura 1.9:** Differenze tra Git e SVN. URL: `https://bit.ly/2ovUUOL`

I repository contenenti tutti i progetti dell'azienda sono ospitati da un server Git privato, installato in uno dei server di WebPD. Tale scelta è stata fatta per essere quanto più tolleranti possibile ad un'eventuale assenza di connessione internet.

### Ambiente di sviluppo

In azienda viene permesso a ciascun programmatore, in base alle sue preferenze, di scegliere l'ambiente di sviluppo che preferisce. In ogni caso ci sono degli strumenti generalmente preferiti, in base alla tecnologia utilizzata per sviluppare

### Sviluppo Web

Per quanto riguarda lo sviluppo web, lo strumento più frequentemente utilizzato è *Visual Studio Code* (la cui *UI* è mostrata in Figura 1.10), un tool multi piattaforma

sviluppato da Microsoft, che si pone in mezzo tra un IDE e un semplice editor di testo. Di base, infatti, non fornisce altro che la funzionalità di *syntax highlight*, ma grazie a innumerevoli plugin, può acquisire, ad esempio, funzionalità avanzate di analisi statica. Inoltre si integra molto bene con Git, integrando degli strumenti per effettuare commit, push e merge.

### Sviluppo Mobile

Dato che viene utilizzato *Cordova*, le tecnologie utilizzate sono (quasi) analoghe a quelle utilizzate nello sviluppo web. Pertanto, anche in questo campo, *Visual Studio Code*, coaudivato dagli adeguati plugin, rimane una scelta consueta. In alternativa, viene usato anche *Jetbrains WebStorm*, un vero e proprio IDE che fornisce funzionalità avanzate di refactoring del codice e di linting.

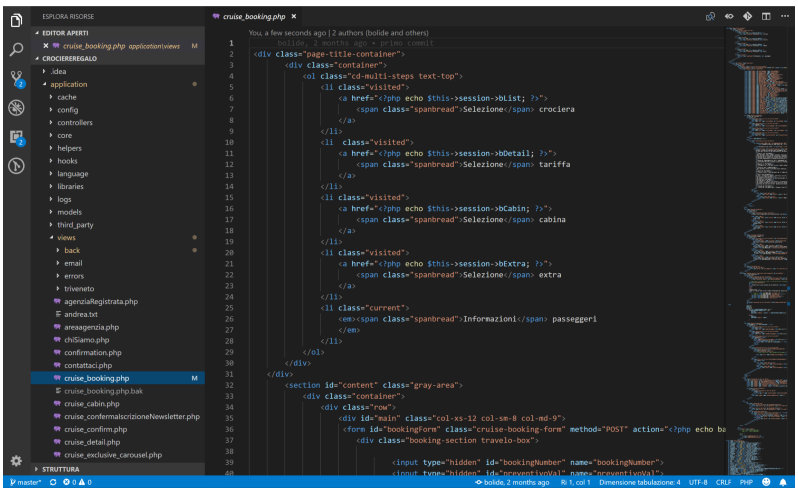

**Figura 1.10:** Schermata di Visual Studio Code

### Sviluppo Desktop

Nello sviluppo di applicazioni desktop, bisogna fare una distinzione in base al linguaggio utilizzato. In caso si tratti di Java, l'IDE più comune è IntelliJ Idea, che offre strumenti di analisi "on-the-fly" e completamento e generazione automatica del codice. Discorso a parte per Delphi, in quanto, essendo una tecnologia proprietaria, si è vincolati all'IDE *Embarcadero RAD Studio* che, tra le altre cose, ha un costo di licenza molto elevato (arriva a sforare i 6000 euro).

## 1.4 Clientela

La tipologia di cliente più comune per WebPD è la piccola-media impresa che vuole realizzare o ammodernare il suo sito web, vetrina o e-commerce che sia. Tale clientela proviene da numerosi settori, quali (per citarne alcuni) edile, farmaceutico, navale e turistico. WebPD possiede numerose soluzioni a portfolio, utili al cliente per capire meglio le sue esigenze, che, molto spesso, non sono ben chiare neanche a lui. Tutti i prodotti offerti, inoltre, sono altamente personalizzabili, in modo da concretizzare fedelmente quanto richiesto dal cliente.

# Capitolo 2

# Descrizione dello stage

## 2.1 Vantaggi per l'azienda

WebPD trae diversi vantaggi dall'attività di stage curricolare che è stata disposta ad ospitare.

Primo su tutti, l'inserimento in azienda, seppur solo per un paio di mesi, di un nuovo membro del personale, mai entrato in contatto con l'azienda. Ciò, in primis, ha permesso di distribuire il carico di lavoro tra più persone, permettendo di accelerare lo sviluppo sui progetti in cantiere. Inoltre, l'introduzione nel team di una persona completamente esterna all'azienda, ha portato un ulteriore punto di vista all'interno del team di sviluppo. Tale punto di vista si è dimostrato utile nel tentativo di risoluzione di alcuni problemi software "cronici" (come la lentezza di esecuzione delle query, problema che verrà descritto nel dettaglio nel prossimo capitolo), permettendo un ragionamento fuori dagli schemi mentali dell'ideatore di tale software.

In secondo luogo, ha permesso all'azienda di esplorare nuovi canoni stilistici per alcuni suoi prodotti a costo zero, come nel caso del restyling della homepage del sito CrociereRegalo (descritta anch'essa nel prossimo capitolo), senza quindi il rischio di sacrificare il lavoro (e la retribuzione) di un membro del personale.

## 2.2 Presentazione del progetto

L'obiettivo di questo stage è permettere a WebPD di completare la riscrittura del sito CrociereRegalo.it. Tale sito, infatti, prima dell'ingresso di Primarete tra le quote di WebPD, era stato realizzato e mantenuto da WebCola, una web agency con la quale Primarete aveva stretto una partnership commerciale, che si è appunto interrotta nel 2015.
Secondo gli accordi presi, il sorgente del sito era di proprietà di WebCola, pertanto non è stato possibile per WebPD procedere ad una semplice modifica/aggiornamento di qualcosa già esistente. Il vecchio sito (il cui aspetto è mostrato in Figura 2.1), inoltre, non era *responsive* e, dai dati di Google Analytics è emerso che la maggior parte delle visite avveniva da dispositivi mobili.
La problematica maggiore, comunque, si sono rivelati gli accordi presi con le varie compagnie di crociera (MSC, Costa, Royal Caribbean, Celebrity e Azamara): tali

accordi, infatti, erano stati presi da WebCola in nome e per conto suo, quindi WebPD si è vista obbligata a ristabilirli.


**Figura 2.1:** Screenshot della versione di CrociereRegalo sviluppata da WebCola

Dal 2015 fino ad luglio 2018, WebPD è riuscita a creare un *Booking Engine* che interagisse con API e WebServices forniti da Costa e MSC, permettendo di acquistare una crociera, pagando tramite un *payment gateway* fornito dal consorzio TVB (Triveneto Bassilichi). La soluzione realizzata, però, aveva grossi problemi di prestazioni (ad esempio la homepage del sito aveva un tempo di risposta medio di 7 secondi, a cui poi doveva essere sommato il tempo di download della risposta, rendering grafico ed esecuzione del codice Javascript presente) e mancava di alcune funzionalità, come la possibilità di vendere tariffe *vuoto per pieno*. Esse non sono altro che posti (cabine) acquistati da un'agenzia viaggi (nel caso in esame, Primarete) ad un prezzo scontato e rivenduti poi ai privati.
Lo stage ha come obiettivo il completamento e l'ampliamento delle funzionalità offerte dal *Booking Engine* alla base di CrociereRegalo. Più nello specifico, lo stage si propone di svolgere le attività descritte in seguito.

### 2.2.1 Studio di funzionamento del *Booking Engine*

La parte introduttiva dello stage si pone come obiettivo quello di analizzare e capire il funzionamento del *Booking Engine*. Infatti quest'ultimo è realizzato tramite il framework *Codeigniter*, e, quindi, gode di una struttura delle classi particolare, come anche il modo in cui vengono gestiti gli URL. Inoltre, usando come DBMS il software *Microsoft SQL Server*, il linguaggio utilizzato per le interazioni con il database è *Transact SQL*, un dialetto di *SQL*, con il quale è necessario prendere confidenza.

### 2.2.2 Ottimizzazione delle performance del *Booking Engine*

Come accennato precedentemente, il *Booking Engine* ha grossi problemi di prestazioni. Ciò è dovuto principalmente alla grande mole di dati contenuta nel database, che deve essere interrogato molteplici volte ad ogni richiesta HTTP ricevuta. Gli scopi di questa fase dello stage sono i seguenti:

1. Ottimizzare il database in modo che le interrogazioni risultino più veloci. A tale scopo *SQL Server* fornisce una funzionalità chiamata *Database Engine Tuning Advisor* (mostrata in Figura 2.2), che permette, sottoponendole delle query da analizzare, di suggerire la creazione di eventuali indici, in modo da aumentare le performance delle interrogazioni;

2. Valutare se implementare o meno (e in caso positivo, farlo) un meccanismo di caching dei risultati delle query, in modo da diminuire le interrogazioni al database il più possibile.

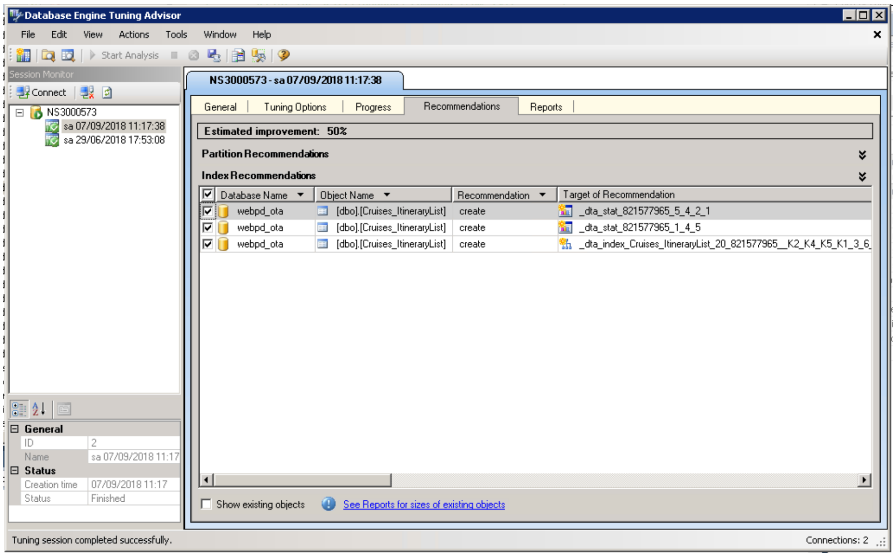


**Figura 2.2:** Screenshot del tool Database Tuning Engine Advisor utilizzato per l'ottimizzazione delle query

### 2.2.3 Integrazione delle tariffe *vuoto per pieno*

Il partner Primarete ha espresso l'esigenza di poter vendere le proprie tariffe *vuoto per pieno* attraverso il *Booking Engine*. Come specificato precedentemente, esse sono dei gruppi di cabine acquistate "al buio" da un'agenzia viaggio ad un prezzo scontato (in quanto ne vengono acquistate più di una in blocco), che poi verranno rivendute dall'agenzia stessa ai clienti. Ovviamente, il guadagno dell'agenzia viaggi è rappresentato dalla differenza tra il prezzo di vendita della cabina e quello di acquisto. Il problema è che le cabine associate al *vuoto per pieno*, agli occhi dei WebServices delle varie compagnie di crociera, risultano prenotate, quindi non disponibili: è necessario modificare il flusso di prenotazione del *Booking Engine* permettendo di inserirvi anche i gruppi di cabine acquistate tramite tariffa *vuoto per pieno*, oltre a quelle ancora

effettivamente disponibili; tale modifica dovrà essere fatta sia a livello *backend* che *frontend*.

### 2.2.4 Aggiunta delle tariffe di un nuovo fornitore

La versione di CrociereRegalo sviluppata da WebCola integrava, oltre alle crociere di MSC e Costa, anche quelle di Royal Caribbean (e controllate, ovvero Celebrity e Azamara). Lo stage, quindi, si prefigge di ripristinare questa funzionalità anche sulla nuova versione del *Booking Engine* CrociereRegalo reintroducendo nel flusso di prenotazione le crociere del gruppo Royal, tramite l'integrazione dei WebServices forniti da IST (Fibos). Tale integrazione comporterà modifiche sia alla parte *frontend* che alla parte *backend* del *Booking Engine*

### 2.2.5 Altri interventi minori

La fase conclusiva dello stage si prefigge di svolgere interventi minori su CrociereRegalo, per migliorarne la SEO ed in generale l'accessibilità, soprattutto dai dispositivi mobili.

## 2.3 Obiettivi dello stage

In fase di definizione contenutistica dello stage, i punti sopra descritti sono stati distribuiti in obiettivi aventi tre livelli di priorità, tenendo conto anche del numero di ore ridotto (circa 310) a disposizione dello stagista, identificati dalle seguenti sigle:

* **Ob** per gli obiettivi obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
* **D** per gli obiettivi desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
* **Op** per gli obiettivi opzionali, rappresentanti valore aggiunto non strettamente competitivo.

| | |
|---|---|
| | **Obbligatori** |
| Ob1 | Interazione con il database SQL Server attraverso le librerie del framework Codeigniter |
| Ob2 | Realizzazione integrazione flat-file di un nuovo fornitore con il Data Exchange del *Booking Engine* |
| Ob3 | Aggiunta prodotti e tariffe del nuovo fornitore ai risultati della ricerca lato *frontend* del *Booking Engine* |
| Ob4 | Esecuzione test e redazione documentazione sul lavoro svolto |
| | **Desiderabili** |
| D1 | Realizzazione del registro carichi/scarichi tariffe "vuoto per pieno" come funzionalità lato *backend* del *Booking Engine* |
| D2 | Interrogazione web-service in tempo reale per sincronizzare prezzi e disponibilità del nuovo fornitore con il Data Exchange |
| D3 | Realizzazione conferma prenotazione al fornitore come funzionalità lato *frontend* del *Booking Engine* |
| | **Opzionali** |
| Op1 | Analisi e realizzazione di nuove funzionalità |

## 2.4 Vincoli

### 2.4.1 Vincoli metodologici

In accordo con il tutor aziendale, lo stage si è svolto presso la sede dell'azienda. Questa scelta è stata fatta principalmente per due motivi:

* agevolare la comprensione, da parte dello stagista, delle dinamiche aziendali e l'interazione con il proponente del progetto CrociereRegalo (WebPD e Primarete hanno l'ufficio all'interno dello stesso palazzo);
* favorire al massimo l'interazione tra stagista e tutor aziendale, evitando ritardi di risposta, problematica che invece può avere il *remote-working*.

Inoltre è stato deciso che, al raggiungimento di ogni obiettivo prefissato, lo stagista avrebbe dovuto redarre una breve relazione, descrivendo le problematiche affrontate, le scelte adoperate e il risultato ottenuto. Tali relazioni, poi, fungeranno da materiale ausiliario per la presentazione delle nuove funzionalità al proponente.
Infine, è stato posto come obbligo che tutto il lavoro prodotto dallo stagista sia sottoposto a versionamento, quindi caricato in un repository Git dedicato, ospitato sul server aziendale.

### 2.4.2 Vincoli temporali

Lo stage ha una durata prevista di *310* ore complessive, distribuite in 9 settimane da 34 ore lavorative ciascuna (ad esclusione della prima, da 38 ore). L'orario di lavoro concordato con il tutor aziendale è stato dal Lunedì al Giovedì dalle 09:00 alle 18:30, con 1 ora di pausa pranzo.
Prima dell'inizio dello stage è stata definita, nel piano di lavoro, una scansione temporale delle attività con granularità settimanale così definita:

* **Prima settimana**: formazione sulle tecnologie utilizzate dal *Booking Engine* CrociereRegalo, con particolare attenzione alla libreria *Codeigniter* e al DBMS *Microsoft SQL Server*;
* **Seconda settimana**: Proseguimento delle attività di formazione iniziate la prima settimana. Ottimizzazione del database tramite gli strumenti forniti da SQL Server;
* **Terza settimana**: Ottimizzazione del sito tramite implementazione della cache delle query. Studio e progettazione dell'aggiunta delle tariffe *vuoto per pieno* al flusso dati del *Booking Engine*;
* **Quarta settimana**: Realizzazione, test e redazione di documentazione di quanto progettato la settimana precedente;
* **Quinta settimana**: Studio e progettazione dell'integrazione dati forniti dal sistema FIBOS (Royal);
* **Sesta settimana**: Realizzazione dell'integrazione di navi, porti, itinerari, categorie di cabina e prezzi attraverso interrogazioni schedulate ai WebServices FIBOS;
* **Settima settimana**: Realizzazione, test e redazione di documentazione di quanto progettato la settimana precedente;

* **Ottava settimana**: Studio e progettazione dell'aggiunta delle tariffe Royal al flusso di prenotazione;
* **Nona settimana**: Realizzazione, test e redazione di documentazione inerente a quanto progettato la settimana precedente.

### 2.4.3 Vincoli tecnologici

A livello implementativo, l'azienda non ha imposto precisi vincoli, se non quello di aderire il quanto più possibile ai paradigmi di programmazione già esistenti ed applicare del buon senso a quanto si intende realizzare. Ciò principalmente significa dovrà essere adottato un design pattern MVC (il cui schema di funzionamento è mostrato in Figura 2.3), intrinseco di *Codeigniter*, e cercare di creare del codice che sia quanto più manutenibile possibile. Ovviamente, quanto realizzato avrebbe dovuto girare correttamente sull'ambiente di esecuzione già utilizzato, ovvero *PHP 7.1* su *Windows Server 2008 R2*, *IIS* come web server e *SQL Server 2012* come *DBMS*.

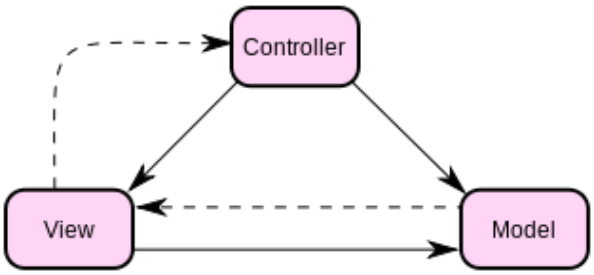


**Figura 2.3:** Schema UML dell'architettura MVC. URL: `https://bit.ly/2wPfE8E`

### 2.4.4 La mia scelta

È da quando ero alle scuole elementari che ho la passione per l'informatica, ed è proprio per coltivare questa passione che ho scelto il cammino di studi che sto portando a termine in questo momento. Sono sempre stato affascinato dalla programmazione web, e il primo linguaggio in cui ho imparato a programmare (da autodidatta, nell'ormai lontano 2012) è stato *PHP*.
Dal 2012 ad oggi ho svolto numerosi progetti utilizzando l'accoppiata HTML/PHP, coadiuvata da Javascript e SQL (MySQL). Molti di questi progetti sono stati solo "esercizi di stile", utili per mettermi alla prova, per mettere alla prova quanto imparato e per cercare di ampliare più possibile le mie conoscenze ed abilità nel campo della programmazione. Alcuni progetti, però, sono anche stati adottati e utilizzati da aziende, il ché mi ha regalato non poche soddisfazioni. In ogni caso, ho sempre fatto il programmatore a livello amatoriale.
Ho scelto di svolgere questo stage per inserire un nuovo tassello nel percorso di crescita personale che sto intraprendendo: volevo confrontare il mio modo "amatoriale" di lavorare con una way-of-working aziendale. Essendo io già abbastanza familiare con le tecnologie utilizzate da *WebPD*, avrei potuto effettuare un confronto molto diretto tra la metodologia di lavoro aziendale e la mia.

# Capitolo 3

# Resoconto dello stage

*In questo capitolo verranno descritte le attività svolte durante lo stage. Per ogni attività si cercherà di descrivere il problema affrontato, le scelte effettuate ed i test svolti.*

## 3.1 Scelte tecnologiche e strumenti utilizzati

A livello tecnologico, dato che il *Booking Engine* è basato sullo stack che in azienda viene chiamato WISP, sono stato obbligato ad utilizzare *PHP* su framework *Codeigniter* per la parte *backend*, *HTML5*, *CSS3*, *Javascript* (principalmente utilizzando jQuery) per quanto riguarda il *frontend*. Infine, per la comunicazione tra *backend* e *frontend* ho adoperato richieste AJAX con JSON come formato per lo scambio di dati. Come ambiente di sviluppo ho deciso di usare *Visual Studio Code* per via della sua semplicità e possibilità di personalizzazione. Infine, tutto il lavoro che ho svolto è stato sottoposto a versionamento utilizzando il server *Git* dell'azienda.

## 3.2 Funzionamento e struttura del *Booking Engine*

Appena iniziato lo stage, mi sono subito dedicato all'analisi della struttura di Crociere-Regalo, grazie anche (soprattutto all'inizio) all'aiuto del mio tutor.

### 3.2.1 Funzionalità

CrociereRegalo è un motore di ricerca di crociere. Permette di trovare una determinata crociera utilizzando dei filtri di ricerca per area geografica (ad esempio Caraibi, Mediterraneo, Nord Europa), per data di partenza con granularità mensile (ad esempio Settembre 2018), per intervalli di durata (da 1-6, 7-8, 9-12 o più di 12 giorni) e per compagnia di crociera (ad esempio MSC Crociere). Per capire bene il suo funzionamento è stato necessario apprendere alcuni termini tecnici affini all'ambiente croceristico, come:

* **Itinerario**: definisce il percorso che fa una crociera (ad esempio "Barcellona, Ajaccio, Civitavecchia, Barcellona"). Nell'arco di un periodo di tempo vi possono essere molteplici crociere percorrenti un singolo itinerario. Ogni itinerario è identificato da un codice che lo contraddistingue univocamente, e la durata dell'itinerario è una proprietà intrinseca (ovvero, all'interno di una compagnia di

crociera, non esistono due itinerari che svolgono lo stesso percorso mettendoci tempi diversi). Ciascun itinerario ha 0 o più partenze nell'arco di un anno;

* **Cabina**: una stanza di una nave da crociera. Esistono varie categorie di cabina, differenziate in base alla grandezza, al posizionamento (ad esempio le cabine interne alla nave, senza quindi finestre, che sono quelle più economiche). Quando si effettua una prenotazione, non viene prenotato un posto letto ma viene prenotata un'intera cabina;

* **Opzione**: consiste nel blocco del prezzo di una cabina per un periodo di tempo che varia in base al fornitore, ma che di solito si attesta tra le 24 e le 72 ore. Tale prezzo infatti, analogamente per quanto avviene con i biglietti aerei, aumenta all'aumentare delle prenotazioni (quindi al passare del tempo).

CrociereRegalo permette di selezionare un itinerario, vederne le partenze, categorie di cabina disponibili e prezzi e poi prenotare od opzionare una cabina.

### 3.2.2 OTA e DataExchange

Mi è stato spiegato che il *Booking Engine*, in realtà, era spezzato in due parti dipendenti l'una dall'altra: **OTA** (disponibile all'indirizzo [4]) e **DataExchange** (disponibile all'indirizzo [5]). L'idea alla base di questa divisione è che la parte *OTA* rappresenti il sito web vero e proprio, con il quale l'internauta si affaccia, mentre la parte *DataExchange* serva per l'interazione tra *OTA* e WebServices dei vari fornitori (dove per fornitori si intendono le varie compagnie di crociera).

### 3.2.3 Interazione tra OTA e DataExchange

*OTA* e *DataExchange* sono dipendenti l'uno dall'altro, ma hanno due database separati. Questo, fondamentalmente, avviene perchè i dati provenienti da i vari fornitori hanno formati diversi, che devono quindi essere uniformati per poter essere processati secondo una logica più indipendente possibile. Il compito del *DataExchange* è proprio questo: interrogare i WebServices dei vari fornitori, ricevere i dati, elaborarli, uniformarli e passarli ad *OTA*.
Vi sono due possibili interazioni tra *OTA* e *DataExchange*

* Interazione **schedulata** (illustrata in Figura 3.1), che avviene circa 3 volte al giorno, il cui compito è sincronizzare i cataloghi (chiamati anche *flatfile*) dei vari fornitori per rendere disponibili le (eventuali) modifiche alla parte *OTA* del *Booking Engine*.
  Quando un visitatore del sito CrociereRegalo cerca una crociera, tale ricerca avviene interrogando i cataloghi presenti nel database della parte *OTA*, senza quindi interrogare i WebServices delle compagnie di crociera (per motivi di prestazione dovuti all'ingente mole di dati da elaborare). I cataloghi, quindi, vengono scaricati nel DataExchange, elaborati (uniformati) e poi sincronizzati con il database di OTA; la procedura di sincronizzazione viene chiamata **Integrazione**. L'integrazione avviene attraverso l'invocazione (grazie allo scheduler di *Windows Server*) che, grazie ad una chiamata *HTTP* ad una particolare pagina del DataExchange, invoca la procedura di integrazione dati.

* Interazione **real-time** (rappresentata in Figura 3.2), che avviene durante tutto il flusso di prenotazione di una cabina (che analizzerò in seguito). Tale flusso

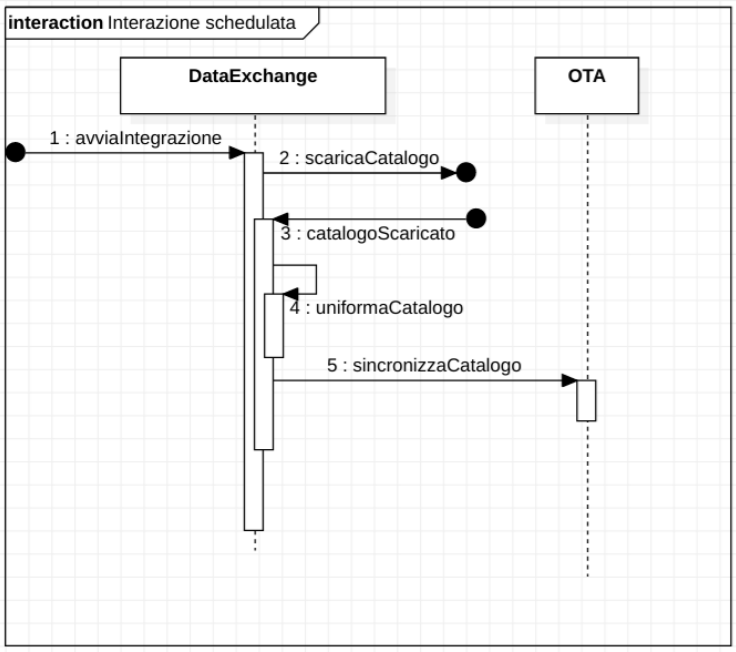


**Figura 3.1:** Schema dell'integrazione schedulata DataExchange-OTA.

deve per forza disporre di dati aggiornati in tempo reale, altrimenti potrebbero verificarsi problemi in fase di prenotazione (come la prenotazione di una cabina non più disponibile). L'interazione in tempo reale avviene attraverso delle chiamate *HTTP* (AJAX) effettuate nel momento del bisogno dall'*OTA* al *DataExchange*, attraverso lo scambio di dati in formato JSON.

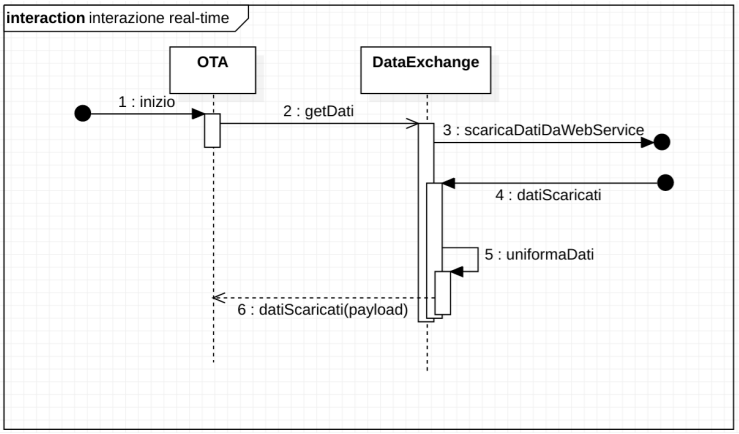


**Figura 3.2:** Schema dell'integrazione in tempo reale DataExchange-OTA.

### 3.2.4 Flusso di prenotazione

Quando il visitatore, in seguito alla ricerca, apre i dettagli di una partenza, viene data la possibilità di poter avviare la procedura (flusso) di prenotazione, che si avvale dell'interazione real-time tra *OTA* e *DataExchange* si compone nei passaggi illustrati in Figura 3.3. Nel dettaglio:

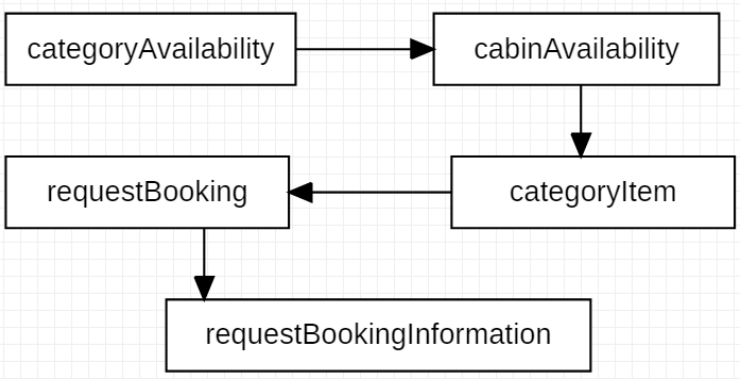


**Figura 3.3:** Schema del flusso di prenotazione

1. *categoryAvailability*: vengono elencati i gruppi (categorie) di cabine effettivamente disponibili (o un messaggio di errore in caso non vi siano più posti prenotabili) tenendo conto del numero di passeggeri inseriti (banalmente, se si selezionano 4 passeggeri, vengono mostrati le categorie di cabine nelle quali vi è presente almeno una quadrupla) e dell'età di tali passeggeri (per il calcolo preciso della tariffa, in quanto un minore paga meno di un adulto);
2. *cabinAvailability*: una volta selezionata una categoria di cabine, vengono mostrate tutte le cabine (una per una) ancora disponibili, raggruppate in base al ponte della nave in cui si trovano;
3. *categoryItem*: vengono elencati tutti i servizi extra (e relativi prezzi) disponibili in aggiunta a quelli già inclusi nel prezzo della cabina, e viene data la possibilità di selezionarli per aggiungerli alla prenotazione;
4. *requestPricing*: viene mostrato il prezzo finale (preso dal WebService del fornitore) della prenotazione, tenendo conto di quanto selezionato negli step precedenti. Tale prezzo include anche le tasse portuali ed eventuali oneri aggiuntivi;
5. *requestBooking*: viene effettuata una prenotazione od un'opzione di quanto selezionato negli step del flusso precedenti. Nel caso di prenotazione, viene anche gestito il pagamento (tramite API fornite dal consorzio *Triveneto Bassilichi*).
6. *requestBookingInformation*: viene mostrato il riepilogo di ciò acquistato, con la possibilità di saldare quanto ancora dovuto (nel caso la prenotazione ammetta il versamento di un acconto).

### 3.2.5 Struttura del codice

Sia *OTA* che *DataExchange* sono realizzati usando il framework *Codeigniter*. Questo implica che applicano a PHP il paradigma *Object-oriented* (orientato agli oggetti)

associato al design pattern MVC. Entrambi i progetti, dunque, presentano tre tipologie di classi:

* **Model**, che estendono la classe base *CI_Model*, i quali sono la chiave di accesso ai dati dell'applicazione. Si è deciso di realizzare un *model* per ogni tabella presente nel database;
* **View**, che generano il codice HTML di ogni pagina (o porzione di essa). Alle *view* è possibile passare delle variabili in modo da rendere dinamico il loro contenuto.
* **Controller**, che estendono la classe base *CI_Controller* e istanziano le *view* popolandole con i dati provenienti dai *model*. La particolarità di queste classi è che i loro metodi pubblici si possono chiamare direttamente tramite URL. Codeigniter, infatti, implementa un meccanismo (grazie al *magic method* $\_\_call$) per il quale è possibile chiamare un metodo pubblico $a$ di una classe controller $C$ semplicemente recandosi all'url $nomedelsito.dominio/C/a$ (ad esempio $crociereregalo.it/C/a$). Tale meccanismo risulta molto utile nella realizzazione di API e nella creazione di *URL* user-friendly (quindi più leggibili)

Codeigniter, inoltre, presenta un modulo personalizzato per interagire con il database, che supporta funzionalità come il *log delle query* (in pratica è possibile sapere l'ultima query eseguita, molto utile in caso di *debug*), meccanismi di *query building* (creazione assitita di query), *query parametrizzate*, gestione delle transazioni ecc.

## 3.3 Ottimizzazione delle performance del database

### 3.3.1 Il problema

Il sito CrociereRegalo soffriva di un grande problema: la lentezza di caricamento delle pagine. Tale lentezza era dovuta principalmente all'elevato numero di query complesse presenti in ogni pagina, ciascuna delle quali aveva dei tempi di esecuzione nell'ordine dei secondi che, complessivamente, facevano sì che il tempo di caricamento di alcune pagine (in primis quella di visualizzazione dei risultati di una ricerca) lambisse pericolosamente il minuto.

### 3.3.2 La soluzione trovata

Dopo un'attenta analisi del database, è emerso che molte tabelle non avevano una corretta struttura. In particolare, su alcune non vi era definita alcuna chiave primaria, mentre in generale non erano presenti chiavi esterne ed indici. Sono quindi state definite chiavi esterne e primarie per tutte le tabelle, mentre per la creazione di indici ci si è affidati all'analisi delle principali query grazie allo strumento *Database Tuning Engine Advisor* integrato nella suite di *SQL Server*. Vengono ora riportati due esempi di analisi svolta e di risultati ottenuti.

#### Query di ricerca

```
SELECT distinct TOP 100 l.Supplier, l.NameIT, l.Description, l.
    Logo, il.ShipCode,il.SailingLengthDays, il.ItineraryCode, po1.
    Code as PortoPartenzaCode ,po1.Name as PortoPartenza, po2.Name
     as PortoArrivo, po2.Code as PortoArrivoCode, min(il.
    SailingDate) as MinSailingDate, min(p.BestPrice) as BestPrice,
    it.GraphicsUrl,sp.Name as ShipName
```

```
FROM Cruises_ItineraryList il inner join Cruises_Lines l on il.
    Supplier= l.Supplier inner join Cruises_Ports po1 on (po1.Code
     = il.DepartingPort and po1.Supplier = il.Supplier ) inner
    join Cruises_Ports po2 on (po2.Code = il.EndPort and po2.
    Supplier = il.Supplier) inner join Cruises_Ship sp on (sp.
    Supplier=il.Supplier and sp.Code = il.ShipCode) inner join
    Cruises_Prices p on (p.Supplier=il.Supplier and p.CruiseId =
    il.CruiseID and p.CruiseCategoryAvailable=1) inner join
    Cruises_Itinerary it on (it.Supplier=il.Supplier and il.
    ItineraryCode = it.Code)
WHERE il.Supplier=1 AND il.AvailabilityStatusCode= 'AV' AND
    BestPrice > 0
GROUP By l.Supplier, l.NameIT, l.Description, l.Logo, il.ShipCode
    ,il.SailingLengthDays, il.ItineraryCode, po1.Code, po2.Code,
    po1.Name, po2.Name , sp.ImgUrl,it.GraphicsUrl,sp.Name order by
     MinSailingDate,PortoArrivo,ItineraryCode
```

Questa è una delle query (la più esterna) che viene usata nel caso di ricerca di un itinerario filtrandolo per fornitore (in questo caso MSC Crociere) i cui esiti vengono utilizzati per creare la schermata presente in Figura 3.4. Il problema di questa interrogazione, oltre alla lunghezza, è l'elevato numero di JOIN tra tabelle aventi un ingente quantitativo di dati, nello specifico:

* Cruises_Prices: 424 780 righe
* Cruises_ItineraryList: 8 092 righe
* Cruises_Ports: 4 187 righe
* Cruises_Itinerary: 2 985 righe

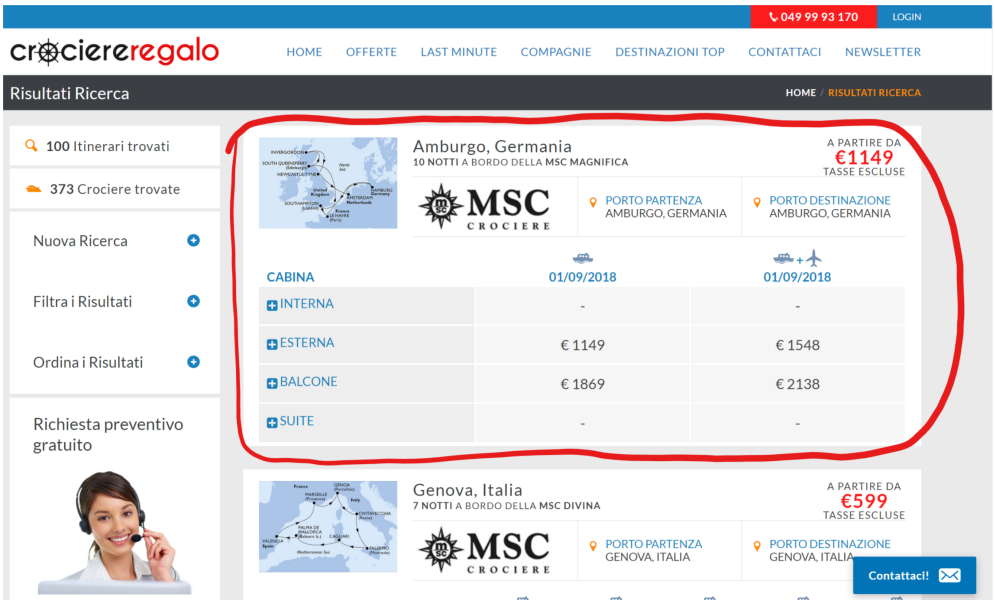


**Figura 3.4:** Esempio di utilizzo del risultato della query della ricerca.

Tali tabelle erano per lo più sprovviste di indici, pertanto le operazioni di JOIN risultavano molto costose. Come risultato si otteneva un tempo di esecuzione, nel server di produzione in assenza di sovraccarichi, variabile tra 23 e 25 secondi. Grazie al tool *Database Tuning Engine Advisor* sono stati creati dei nuovi indici, precisamente su tutte le colonne coinvolte nelle clausole di JOIN. Questo ha portato un ingente riduzione nei tempi di esecuzione della query, che sono arrivati a sfiorare i 2 secondi. Vi è stato, quindi, un miglioramento di ben 21 secondi (circa il 91%) nel tempo di esecuzione della query e di conseguenza nel tempo di risposta della pagina web incaricata di elaborare e visualizzare i risultati della ricerca.

### Query dei last minute

In homepage vi è una sezione che visualizza tutte le offerte *last minute* (ovvero crociere prossime alla partenza aventi ancora posti disponibili a prezzo scontato) presenti nel database, come mostrato in Figura 3.5. Queste ultime vengono inserite a mano dai dipendenti di *Primarete*, ma sono collegate a crociere effettivamente presenti. La query sottostante restituisce tali offerte.

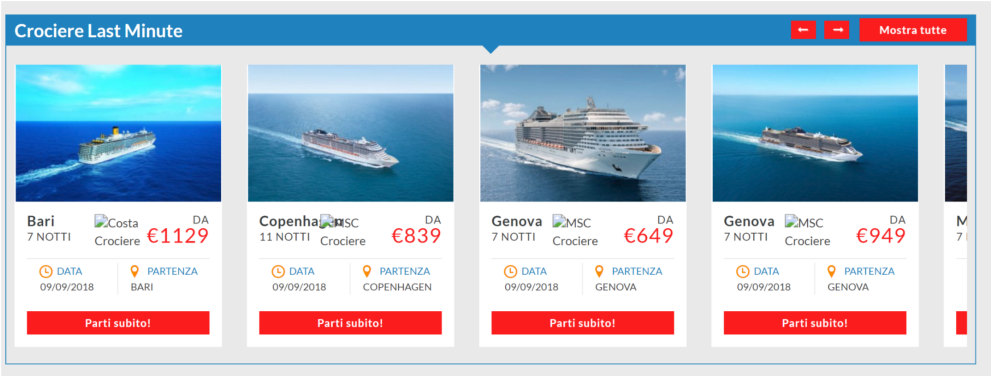


**Figura 3.5:** Esempio di utilizzo del risultato della query dei last minute.

```
SELECT distinct top 4 l.Logo,l.NameIT as SupplierName,il.Supplier
    ,il.CruiseID,il.ShipCode, il.ItineraryCode,il.DepartingPort,il
    .EndPort,il.SailingDate,il.ReturnDate,il.SailingLengthDays as
    NightNumber,po.Name as Partenza,po2.Name as Arrivo,s.foto,sp.
    ImgUrl,min(p.BestPrice) as BestPrice FROM Cruises_
    ItineraryList il join Cruises_Prices p on il.CruiseID = p.
    CruiseId join Cruises_Ports po on (il.DepartingPort = po.Code
    and po.Supplier=il.supplier) join Cruises_Ports po2 on (il.
    EndPort = po2.Code and po2.Supplier=il.supplier) join Cruises_
    Ship sp on (sp.Code=il.ShipCode and sp.Supplier=il.Supplier)
    join Cruises_Lines l on (l.Supplier=il.Supplier) left join
    Cruises_Banner AS b ON b.tipo = 'nave' AND b.riferimento =
    CONCAT(il.Supplier,'-',il.ShipCode) INNER JOIN Cruises_Banner_
    Slide AS s ON b.id = s.banner AND s.nome='nave'
 WHERE SailingDate BETWEEN '2018-09-09' and '2018-10-09' and
     AvailabilityStatusCode='AV' and il.supplier = 2 and po.
     Supplier= 2 and po2.Supplier=2 and p.BestPrice>0 group by l.
     Logo,l.NameIT,il.Supplier,il.CruiseID,il.ShipCode, il.
```

```
    ItineraryCode,il.DepartingPort,il.EndPort,il.SailingDate,il.
    ReturnDate,il.SailingLengthDays,po.Name ,po2.Name ,s.foto,sp.
    ImgUrl
```

Anche qua, il numero di JOIN su tabelle aventi ingenti quantità di dati rende il tempo di esecuzione della query molto elevato in caso il database non sia sufficientemente ottimizzato. Grazie alle ottimizzazioni suggerite dal tool *Database Tuning Engine Advisor*, si è avuto un miglioramento del tempo di esecuzione indicativamente dell'83%, che è passato da 3 secondi a 0.5. Tale miglioramento si è riflettuto anche nelle performance della homepage, che ha visto una netta diminuzione del tempo medio di risposta.

### 3.3.3 Risultati delle ottimizzazioni

Al termine delle ottimizzazioni, sono stati ottenuti buoni risultati (il tempo di risposta del sito è diminuito di almeno il 50%), ma non abbastanza per rendere il sito reattivo ed equiparabile in termini di velocità ai competitor. Si è deciso quindi di esplorare nuove soluzioni.

## 3.4 Cache delle query

### 3.4.1 Il problema

Se è vero che, al termine degli interventi sulla struttura del database descritti nella sezione precedente, le prestazioni delle interrogazioni erano migliorate molto, è anche vero che il sito non aveva raggiunto livelli di prestazioni ottimali. Infatti alcune pagine avevano ancora un tempo di risposta molto elevato. Prendendo come esempio la pagina del sito più utilizzata, ovvero quella di visualizzazione dei risultati di una ricerca, essa è passata da un tempo di risposta tra i 45 e i 60 secondi ad un tempo di risposta tra i 15 e i 20 secondi, comunque troppo per le aspettative di un internauta medio: basti pensare che il principale competitor (`www.logitravel.it`) ha un tempo di risposta che si attesta attorno al secondo e mezzo (10 volte meno) per la stessa funzionalità.

### 3.4.2 La soluzione

La soluzione trovata a tale problema nasce dalla seguente considerazione: è possibile prevedere quando cambia la maggior parte dei dati presenti nel database e utilizzati per le query più lente. Infatti dati inerenti a navi, itinerari, cabine, prezzi e disponibilità vengono rinnovati ad ogni integrazione, che è un'operazione schedulata 4 volte al giorno, ad orari noti. Osservando inoltre che la stessa query sugli stessi dati presenta gli stessi risultati, si è giunti alla conclusione che l'implementazione di un meccanismo di caching delle query permetta di ottimizzare velocità del sito e carico del server.
Il meccanismo progettato ha il seguente funzionamento:

* la query viene eseguita la prima volta ed il suo risultato viene salvato nella cache;
* quando la query viene eseguita ancora, invece di interrogare il database viene letto il contenuto della cache;
* ad ogni integrazione dati la cache viene svuotata, perchè i dati presenti in essa non riflettono più il nuovo contenuto delle tabelle del database;

* al termine di ogni integrazione, vengono effettuate delle richieste *HTTP* alle pagine più utilizzate del sito, in modo tale che la cache venga ricreata.

Si è deciso di affidare la gestione della cache a *Codeigniter*, che offre già questa funzionalità. Il modulo di gestione del database di *Codeigniter*, infatti, fornisce già i seguenti metodi:

* *cache_ on*, che permette di abilitare la cache;
* *cache_ off*, che permette di disabilitare la cache;
* *cache_ delete*, che permette di cancellare la cache di una singola pagina (passata come parametro);
* *cache_ delete_ all*, che permette di cancellare tutto il contenuto della cache.

Combinando l'uso di *cache_ on* e *cache_ off*, si può anche disabilitare la cache per determinate query, che magari operano su dati più "volatili" e aventi minor quantità. In generale, comunque, *Codeigniter* salva i risultati delle varie query in dei file di cache (aventi come nome l'*hash* della query a cui tale file si riferisce) suddivisi per cartelle in base all'url che le ha generate, come mostrato in Figura 3.6.


**Figura 3.6:** Struttura di cartelle e file di cache in esse contenuti.

Grazie a *cache_ delete* sarebbe possibile cancellare la cache solo di alcune pagine, ma è stato preferito non utilizzare tale funzione in quanto avrebbe aumentato inutilmente la complessità della soluzione.

### 3.4.3 Risultati

L'implementazione della cache ha giovato molto al *Booking Engine*. Basti pensare che ora la pagina di visualizzazione dei risultati ricerca è giunta ad avere un tempo

di risposta medio attorno ai 3 secondi, ovvero ora è l'80% più veloce di prima, ma soprattutto è paragonabile al tempo di risposta dei competitor. Il tempo di risposta della homepage, inoltre, è arrivata a toccare i 500 millisecondi, dai circa 3 secondi prima di implementare la cache.
Tra ottimizzazione del database e cache, comunque, sono stati fatti ingenti miglioramenti alle prestazioni, che riassumo nella tabella 3.1, prendendo come esempio la pagina principale e quella di visualizzazione dei risultati della ricerca (che, per brevità, chiamerò soltanto "ricerca").

| **Pagina** | **Senza ottimizzazioni** | **Con DB ottimizzato** | **Con cache** |
|---|---|---|---|
| Homepage | 7 secondi | 3 secondi | 0.5 secondi. |
| Ricerca | 45-60 secondi | 15-20 secondi | 3 secondi. |

**Tabella 3.1:** Risultati delle ottimizzazioni effettuate sul *Booking Engine*

## 3.5 Integrazione delle tariffe *vuoto per pieno*

### 3.5.1 Analisi del problema

Dopo aver preso effettivamente confidenza con il database, mi è stato chiesto di creare un sistema che permettesse di inserire nel flusso di prenotazione anche tariffe *vuoto per pieno*. Precisamente, tale sistema avrebbe dovuto avere le seguenti caratteristiche:

1. Permettere di inserire nel sistema gli acquisti di *vuoto per pieno* effettuati da *Primarete* tramite upload di appositi file;
2. Permettere di differenziare i prezzi in base al soggetto a cui si sta vendendo;
3. Permettere di prenotare e pagare una cabina con tariffa *vuoto per pieno*.

#### Inserimento degli acquisti nel sistema

Il sistema deve prevedere una funzionalità di "carico" delle tariffe. In fase di acquisto di *vuoto per pieno* da parte di *Primarete*, il fornitore (ovvero la compagnia di crociera) rilascia un file riepilogativo contenente il dettaglio di quanto acquistato. Tali informazioni devono potersi caricare in modo assistito nel sistema, che avrebbe dovuto poi inserirle nel flusso di prenotazione.
Il problema principale è stato che il file ritornato da ciascun fornitore aveva (e ha tuttora) il proprio formato dati, quindi si sarebbe dovuto creare un importatore per ogni fornitore. Inoltre, anche la tipologia di dati presente in ciascun file poteva differire: ad esempio, per riferirsi all'età di ogni passeggero, alcuni fornitori utilizzano l'età ed altri la data di nascita. Il sistema avrebbe dunque dovuto uniformare i dati caricati.

#### Differenziazione dei prezzi

CrociereRegalo è un *Booking Engine* utilizzato per vendite sia B2C (cioè verso clienti finali) sia B2B (cioè verso altre agenzie). Queste ultime, in base agli accordi commerciali che hanno con *Primarete*, appartengono a determinati *listini*, ovvero hanno diritto a commissioni/tariffe più o meno agevolate.
Il sistema, quindi, avrebbe dovuto poter differenziare i prezzi di vendita delle cabine in base al listino di appartenenza dell'acquirente (un acquirente non appartenente a nessun listino sarebbe stato identificato al pari di una vendita B2C).

#### Inserimento della tariffa nel flusso di prenotazione

Il *Booking Engine* avrebbe dovuto permettere di visualizzare i prezzi delle cabine soggette a tariffe *vuoto per pieno* nei risultati di ricerca. Inoltre, avrebbe dovuto essere possibile poter prenotare e conseguentemente pagare tali cabine.
La disponibilità delle stesse, quindi, si sarebbe dovuta aggiornare (decrementare) automaticamente.

### 3.5.2 Progettazione e codifica della soluzione

#### Inserimento degli acquisti nel sistema - Progettazione

Affinché fosse possibile inserire gli acquisti nel sistema, si è progettato di realizzare una nuova pagina riepilogativa nel pannello di amministrazione del *Booking Engine*, accessibile solo tramite l'inserimento di username e password. In tale pagina viene

visualizzato un riepilogo dei *vuoto per pieno* caricati con la possibilità, per ognuno di essi, di avere il dettaglio delle cabine vendute e di quelle ancora disponibili. È stato inoltre inserito un tasto che apre una finestra di dialogo all'interno della quale viene data la possibilità di caricare nuovi *vuoto per pieno*, andando quindi a "rifornire" il "magazzino". In base al fornitore selezionato nella finestra di dialogo è possibile:

* Caricare il file di riepilogo di quanto acquistato fornito dalla compagnia, in caso di fornitore **Costa** o **MSC**, come mostrato in Figura 3.7. Il primo fornisce un file con estensione *.xls* mentre il secondo un file con estensione *.csv*

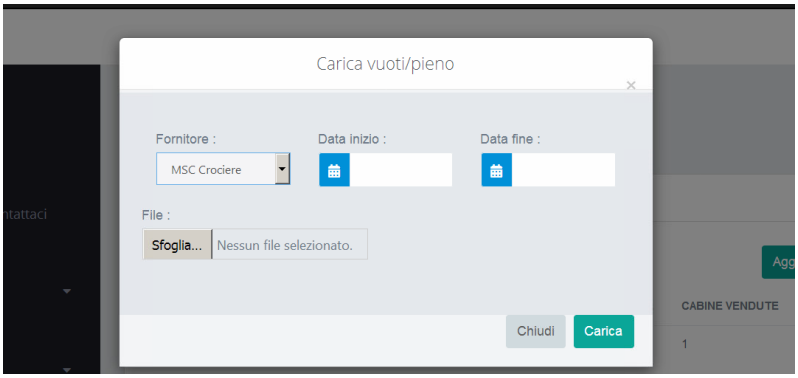


**Figura 3.7:** Schermata di caricamento del vuoto pieno MSC e Costa.

* Inserire a mano i dettagli del *vuoto per pieno* che si intende caricare nel caso di fornitore **Royal Caribbean**, **Celebrity** o **Azamara**, come mostrato in Figura 3.8. Essi infatti forniscono il file di riepilogo di quanto acquistato in formato *.pdf*, che è molto difficile da interpretare.

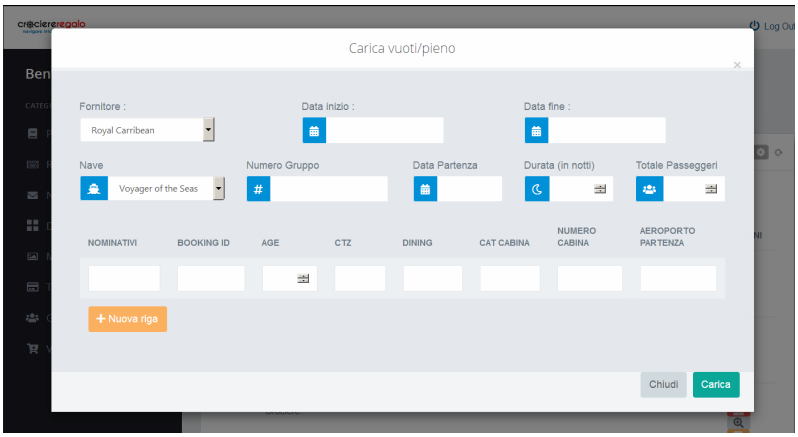


**Figura 3.8:** Schermata di caricamento del *vuoto per pieno* Royal/Celebrity/Azamara.

In ogni caso, è stato deciso che per ciascun *vuoto per pieno* si sarebbero memorizzate le seguenti informazioni:

* Il **fornitore**, ovvero MSC, Costa, Royal Caribbean, Celebrity o Azamara;

* Il **groupID**, ovvero un identificatore univoco per ciascun fornitore del *vuoto per pieno* acquistato;
* Il **cruiseID**, ovvero un identificatore univoco per ciascun fornitore della crociera a cui tale *vuoto per pieno* si riferisce;
* La **data di partenza** della crociera;
* La **lunghezza** (in numero di notti) della crociera;
* Il **numero di posti** (ovvero la capienza massima di persone) totali in quel *vuoto per pieno*;
* Il **numero di cabine** totali di quel *vuoto per pieno*,
* La **data di inizio** e la **data di fine** validità della tariffa, in modo che sia possibile venderla solo in determinati periodi/intervalli. Ciò significa che, ad esempio, se la data di inizio viene impostata al 20 agosto 2018 e la data di fine al 30 settembre 2018, in caso di una ricerca fatta prima del 20 agosto 2018 o dopo il 30 settembre 2018, la tariffa non comparirà tra i risultati.

#### Inserimento degli acquisti nel sistema - Codifica

Oltre alla realizzazione della pagina riepilogativa precedentemente progettata, per memorizzare queste informazioni è stata creata una nuova tabella in database denominata **Cruises_Inventory**, avente come campi *Id* (chiave primaria auto incrementante), *Supplier* (fornitore, chiave esterna che si riferisce a *Cruises_Lines*, tabella dove sono memorizzati tutti i fornitori), *GroupID*, *CruiseID*, *DepartureDate* (data di partenza della crociera), *Length* (lunghezza in notti della crociera), *N_Pax* (numero di posti totali), *N_Cab* (numero di cabine totali), *Data_Inizio* e *Data_Fine*.

Oltre alle informazioni generali relative a ciascun *vuoto per pieno*, è stato necessario memorizzarne anche il contenuto in dettaglio, in termini di cabine acquistate (e quindi da rivendere). Si è deciso di creare la tabella **Cruises_InventoryDetail** avente i seguenti campi:

* **Id**, chiave primaria auto incrementante;
* **Cruises_Inventory**, chiave esterna collegata all'*Id* dell'omonima tabella *Cruises_Inventory*, che serve per collegare ogni riga del dettaglio al *vuoto per pieno* corrispondente;
* **Cruises_PricesID**, chiave esterna collegata all'*Id* di *Cruises_Prices*, tabella il cui scopo verrà analizzato nel dettaglio nella sezione successiva;
* **Cabin**, numero di cabina (univoco nel dominio del fornitore) a cui la riga corrente fa riferimento;
* **BookId**, numero di prenotazione del fornitore della cabina a cui la riga corrente fa riferimento;
* **Category**, categoria della cabina corrente.

Per gestire il caricamento dei *vuoto per pieno* nel sistema sono state realizzate tre classi *parser* (una per MSC, una per Costa e una per il trio Royal-Celebrity-Azamara)in grado di interpretare dei dati in input e renderli compatibili con la struttura delle tabelle *Cruises_Inventory* e *Cruises_InventoryDetail*. Queste tre classi, *MSCParser*, *CostaParser* e *RoyalParser* implementano tutte l'interfaccia **ParserInterface**, che ha i seguenti metodi astratti:

| **Metodo** | **Descrizione** |
|---|---|
| + getCruiseID() : string | Ritorna il CruiseID corrispondente al *vuoto per pieno* considerato |
| + getGroupID() : string | Ritorna il GroupID corrispondente al *vuoto per pieno* considerato |
| + getDepartureDate() : string | Ritorna la data di partenza corrispondente al *vuoto per pieno* considerato, in formato "Y-m-d" (esempio 2018-09-27). |
| + getLength() : int | Ritorna la lunghezza (in numero di notti) corrispondente al *vuoto per pieno* considerato. |
| + getNPax() : int | Ritorna la capienza totale (in numero di persone) corrispondente al *vuoto per pieno* considerato. |
| + getNCab() : int | Ritorna la capienza totale (in numero di cabine) corrispondente al *vuoto per pieno* considerato. |
| + popolaDettaglioTabella() : array | Ritorna un array che, per ciascuna riga del *vuoto per pieno*, associa a ciascun campo della tabella *Cruises_InventoryDetail* il relativo valore. |

È stata poi creata una nuova classe, **Cruises_Inventory**, derivata da *CI_Model* (classe base dei modelli di Codeigniter), che si occupa principalmente del caricamento del *vuoto per pieno* nel database, avvalendosi dei parser prima elencati. Più precisamente, la classe ha i seguenti metodi:

| **Metodo** | **Descrizione** |
|---|---|
| + validId(id:int) : boolean | Ritorna *true* se il parametro passato è un valore presente nella colonna "Id" della tabella *Cruises_Inventory* |
| + loadFromFile (supplier: int, fileContent: string, dataInizio: Date, dataFine: Date) : void | In base al fornitore passato come parametro (*supplier*), carica il contenuto del *vuoto per pieno* (presente in *fileContent*) nel database, avvalendosi del parser associato al fornitore. |

| | |
|---|---|
| - loadRoyalInput (supplier: int, content: array) : boolean | Carica il *vuoto per pieno* Royal/Celebrity/Azamara (in base al valore della variabile *supplier*) leggendo il contenuto dall'array associativo *content* ed elaborandolo grazie al parser *RoyalParser*. |
| - loadCostaFile (fileContent: string) : boolean | Carica il *vuoto per pieno* Costa nel database; *fileContent* rappresenta il contenuto del file del *vuoto per pieno* caricato, che verrà elaborato dal parser *CostaParser*. |
| - loadMSCFile (fileContent: string) : boolean | Carica il *vuoto per pieno* MSC nel database; *fileContent* rappresenta il contenuto del file del *vuoto per pieno* caricato, che verrà elaborato dal parser *MSCParser*. |
| - saveParsedFile(parser: ParserInterface, supplier: int) : boolean | Metodo che viene chiamato da *loadRoyalInput*, *loadCostaFile*, *loadMSCFile*; serve a popolare la tabella *Cruises_InventoryDetail* e la tabella *Cruises_Prices*. |

La gestione del pannello di amministrazione è affidata ad un unico *controller*, chiamato **Back**. In questa classe è stato aggiunto un metodo chiamato *caricaVuotoPieno* che prende l'input dal *frontend* e lo passa al metodo *loadFromFile* della classe *Cruises_Inventory* appena descritta, che si occuperà dell'elaborazione e della restituzione del risultato della stessa sotto forma di JSON.

### Differenziazione dei prezzi - Progettazione

I prezzi di ogni cabina presente nel sistema risiedono nella tabella *Cruises_Prices*. Tale tabella permette di conoscere il prezzo di ogni categoria di cabina (per ogni combinazione di età dei passeggeri tollerata dal sistema) per ogni tariffa disponibile. Il problema è che *Cruises_Prices* viene svuotata e ripopolata ad ogni integrazione: per questo motivo si è deciso di separare la gestione dei prezzi delle cabine soggette a tariffa *vuoto per pieno* in una nuova tabella, che è stata chiamata *Cruises_InventoryPrices*.

### Differenziazione dei prezzi - Codifica

Per realizzare quanto progettato, sono stati fatti i seguenti interventi:

* Nella tabella *Cruises_Prices* è stata inserita una nuova colonna di tipo intero, **Manuale**, che indica se la riga si sta riferendo ad una tariffa passata dal fornitore o presa da un *vuoto per pieno*. Quando viene fatta un'integrazione, vengono eliminate solo le righe aventi valore di tale colonna pari a 0, in modo da non perdere le informazioni relative al *vuoto per pieno*;
* È stata creata la tabella **Cruises_InventoryPrices**, avente i seguenti campi:
  - *Id*, identificatore univoco della riga;

- *Cruises_Inventory*, chiave esterna che si riferisce al campo *Id* di *Cruises_Inventory* e specifica il *vuoto per pieno* di appartenenza di tale riga;
- *Listino*, chiave esterna di *Cruises_Listini*, tabella appena creata che verrà descrittà nel prossimo paragrafo. In caso di valori NULL, il listino si intende essere *B2C*, ovvero vendita al dettaglio;
- *Categoria*, che rappresenta la categoria della cabina a cui ci si sta riferendo;
- *Cabina*, che rappresenta il numero di cabina a cui ci si sta riferendo;
- *BookId*, che indica il codice di prenotazione del fornitore collegato alla cabina a cui ci si sta riferendo;
- Un campo per ogni combinazione di prezzi tollerata dal sistema, ovvero *Price1Adult, Price2Adult, Price3Adult, Price4Adult, Price1Adult1Junior, Price1Adult1Junior1Child, Price2Adult1Child, Price2Adult2Child, Price2Adult1Junior, Price2Adult2Junior, Price2Adult1Junior1Child*;
- *Commissione*, che quantifica la commissione spettante all'agenzia viaggi collegata al listino specificato, in caso il campo *Listino* non sia NULL.

La tabella **Cruises_Listini** menzionata precedentemente è stata creata per poter gestire tutti i listini da collegare alle varie agenzie viaggi. Tale tabella presenta due campi, *Id* e *Nome*, usati appunto per definire il listino stesso.
Nella tabella *Cruises_Utenti*, che raccoglie tutte le agenzie viaggio che possono accedere alla modalità di acquisto B2B, è stato aggiunto il campo *Listino*, chiave esterna di *Cruises_Listini*, in modo da associare ogni agenzia registrata ad un listino, in base agli accordi presi con *Primarete*.

Per l'interazione con la tabella *Cruises_InventoryPrices* è stato creato un nuovo model avente lo stesso nome della tabella, con i seguenti metodi:

| **Metodo** | **Descrizione** |
|---|---|
| + getColonnePrezzi() : array | Ritorna un array contentente la lista di tutte le combinazioni di prezzi tollerate dal sistema. Questo metodo è stato scritto per rendere poi dinamico l'inserimento dei prezzi in questa tabella, ovvero in caso si voglia aggiungere un nuovo campo non presente (ad esempio *Price6Adult*), sarà sufficiente aggiungerlo come colonna della tabella, e interfaccia grafica e query di inserimento si adatteranno automaticamente. |
| + tentaAggiornamentoPrezzi (tabella: array, inventoryId: int) : void | Metodo che prova ad aggiornare i prezzi nel caso di caricamento di un *vuoto per pieno* contenente cabine già presenti nel sistema. |
| + salvaPrezzi (input: array) : boolean | Metodo che salva i prezzi di ogni cabina, per ogni listino, presenti nel parametro *input*. Ritorna *true* in caso di successo, *false* altrimenti. |

| | |
|---|---|
| + modificaPrezzi (input: array) : boolean | Metodo che modifica i prezzi di ogni cabina, per ogni listino, presenti nel parametro *input*. Ritorna *true* in caso di successo, *false* altrimenti. |

È stato inoltre creato un metodo *salvaPrezzi* nel controller *Back*, che si limita ad invocare il metodo *salvaPrezzi* del model *Cruises_InventoryPrices* con i parametri passati tramite POST.
A livello *frontend*, si è fatto in modo che, al termine del caricamento del *vuoto per pieno*, si venga reindirizzati ad una pagina di inserimento prezzi (come quella mostrata in Figura 3.9), che permette appunto di specificare ogni combinazione di prezzo tollerata dal sistema per ogni categoria di cabina per ciascun listino.

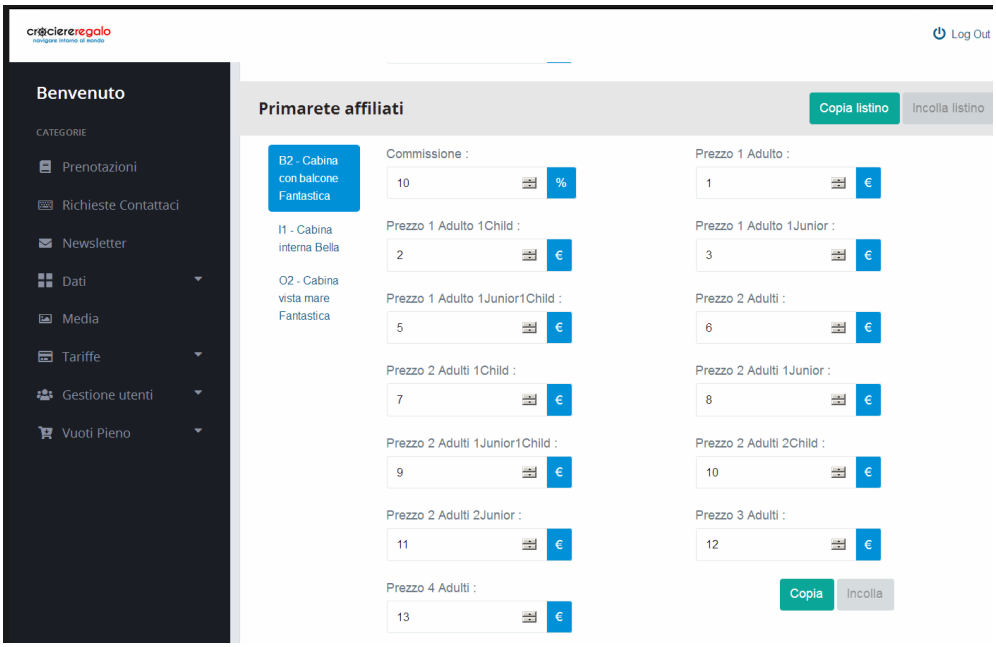


**Figura 3.9:** Schermata di inserimento dei prezzi di un *vuoto per pieno*.

I prezzi poi vengono salvati tramite una chiamata AJAX che invoca il metodo *salvaPrezzi* del controller *Back* precedentemente descritto.

### Inserimento della tariffa nel flusso di prenotazione - Progettazione

Come già accennato nella Sezione 3.2.4, il flusso di prenotazione si compone di 6 step, a ciascuno dei quali corrisponde un metodo nel controller **WS_Cruises** del *dataExchange*. Questi 6 metodi venivano (e vengono tuttora) chiamati tramite richieste AJAX dal *frontend* e al loro interno chiamavano l'omologo metodo in grado di interfacciarsi con i WebService del fornitore considerato. Si è deciso quindi di creare 6 nuovi metodi, uno per ogni step del flusso che, invece di interfacciarsi con qualche WebService, leggesse i

dati delle tariffe *vuoto per pieno* direttamente dal database.

#### Inserimento della tariffa nel flusso di prenotazione - Codifica

Sono stati quindi creati i seguenti metodi:

| Metodo | Descrizione |
|---|---|
| + Inventory_categoryAvailabilty (inout wsResult:array) : void | Concatena al risultato eventualmente restituito dai WebService del fornitore le categorie di cabine, se presenti, aventi tariffa *vuoto per pieno*. Il codice della tariffa restituita (dato che servirà alle successive funzioni) è "INVENTORY". |
| + Inventory_cabinAvailabilty () : array | Legge dal database (facendo delle query su Cruises_Inventory, Cruises_InventoryDetail e Cruises_InventoryPrices) e restituisce le singole cabine disponibili, compreso il prezzo, secondo i qualificatori che legge dal *payload* della richiesta, quali numero ed età dei passeggeri, cruiseID considerato, categoria delle cabine di cui si vuole controllare la disponibilità e listino di appartenenza dell'utente. |
| + Inventory_requestPricing () : array | Calcola un preventivo in base ai parametri passati nel *payload* della richiesta, ovvero fornitore, cruiseID, numero della cabina, dati anagrafici dei passeggeri. |
| + Inventory_requestBooking () : string | Prenota la cabina passata come parametro nel *payload* della richiesta e ritorna il bookingID (identificativo della prenotazione) corrispondente. |
| + Inventory _requestBookingInformation (bookId: string) : array | Dato il bookId passato come parametro, restituisce tutti i dettagli inerenti alla prenotazione. |

Il primo metodo del flusso, *categoryAvailability*, in base al valore del parametro *supplier* (letto dal *payload* della richiesta), invocava i metodi *MSC_ categoryAvailability* o *Costa_ categoryAvailability*, che interrogano i rispettivi WebServices. È stato sufficiente quindi che il nuovo metodo creato *Inventory_ categoryAvailabilty* venisse invocato da *categoryAvailabilty* a prescindere dal *supplier*, concatenando all'output dei metodi *MSC_ categoryAvailability* o *Costa_ categoryAvailability* le categorie di cabine eventualmente disponibili con tariffa *vuoto per pieno*.
Nei metodi *cabinAvailability*, *requestPricing*, *requestBooking* e *requestBookingInformation*, è stata introdotta un'istruzione condizionale che, in base al **fareCode** (codice della tariffa) passato come parametro, decide se invocare l'omologo metodo che legga

dal database delle tariffe *vuoto per pieno* (in caso di fareCode = "INVENTORY") o dai WebServices del fornitore.
Al *frontend*, infine, non è stato necessario fare alcuna modifica.

### 3.5.3 Test

Al termine della realizzazione, sono stati effettuati test manuali sulla congruenza dei dati tra le nuove tabelle ed i *vuoto per pieno* importati. Nello specifico:

* Per i *vuoto per pieno* **MSC** e **Costa**, si è verificato che il file venisse correttamente importato, interpretato e salvato nelle nuove tabelle realizzate *Cruises_ Inventory* e *Cruises_ InventoryDetail*;
* Per i *vuoto per pieno* **Royal**, **Celebrity** e **Azamara** si è verificato che la schermata di caricamento funzionasse e i dati venissero correttamente salvati nelle tabelle *Cruises_ Inventory* e *Cruises_ InventoryDetail*.

Infine si è verificato che i dati presenti nelle tabelle fossero interpretati in maniera adeguata dalle varie query presenti nel flusso di prenotazione, dal primo all'ultimo dei sei step. Nello specifico:

* Si è verificato che il metodo **Inventory_categoryAvailabilty** restituisse correttamente le categorie di cabine di un *vuoto per pieno* preventivamente importato;
* Si è verificato che il metodo **Inventory_cabinAvailabilty** restituisse correttamente le cabine di un *vuoto per pieno* preventivamente importato;
* Si è verificato che il metodo **Inventory_requestPricing** restituisse un preventivo corretto per una cabina disponibile di un *vuoto per pieno* preventivamente importato, tenendo conto correttamente dei qualificatori passati (come l'età dei passeggeri);
* Si è verificato che il metodo **Inventory_requestBooking** effettuasse e salvasse correttamente una nuova prenotazione a database, restituendone il codice corretto;
* Si è verificato che il metodo **Inventory_requestBookingInformation** restituisse i dettagli corretti di un *vuoto per pieno* preventivamente prenotato, in base a quanto presente nel database.

## 3.6 Aggiunta delle tariffe di un nuovo fornitore

### 3.6.1 Analisi del problema

L'ultimo argomento affrontato durante lo stage è stato aggiungere la possibilità di prenotare una crociera con *Royal Caribbean*, *Celebrity* e *Azamara*. Queste tre compagnie forniscono, grazie alla società *ISTINFOR*, dei WebServices chiamati **FIBOS**, che sono unici per le tre compagnie di crociera sopra citate.
*FIBOS* prevede l'utilizzo di due tipologie di WebServices: **FDF** (Fibos Data Feed) e **RCCL** (Royal Caribbean Cruise Line).

#### Importazione delle informazioni statiche

Per potersi interfacciare con i WebServices, è stato necessario aggiungere al *Booking Engine* le informazioni (codici, descrizioni) di aree geografiche, navi, porti e categorie di cabine utilizzate da FIBOS. Queste informazioni sono considerate statiche in quanto non cambiano spesso: si possono verificare delle variazioni solo in caso di varo di una nuova nave e/o costruzione di un nuovo porto.

#### Utilizzo di FDF

Questi WebServices sono progettati per il trasferimento di grandi quantità di dati e forniscono un sistema di ricerca di crociere (e relativi prezzi). Si sono quindi dimostrati adatti al meccanismo di integrazione schedulata presente nel *Booking Engine*. È stato dunque necessario creare un sistema che interrogasse i WebServices *FDF*, consolidandone i risultati nel *DataExchange* ed integrandoli poi nell'*OTA*.

#### Utilizzo di RCCL

Per far fronte ai dati richiesti in tempo reale dal flusso di prenotazione, invece, sono stati offerti da *FIBOS* i WebServices **RCCL**, che è stato necessario integrare ed interrogare.

### 3.6.2 Progettazione e codifica della soluzione

#### Importazione delle informazioni statiche - Progettazione

Informazioni circa codici di aree geografiche, navi, porti, categorie di cabine, ponti ci sono state fornite da *ISTINFOR* in un file formato *.xlsx*. In caso di variazioni di tali informazioni, per contratto, ci sarebbe stato mandato un nuovo file aggiornato. Proprio per questo è stato deciso, invece di importare a mano i dati, di realizzare un importatore.
A livello grafico, si è deciso di realizzare una semplice finestra di dialogo che permettesse di caricare il file in oggetto e mandarlo al *backend* per l'elaborazione. A livello *backend*, si è deciso di realizzare una nuova classe che gestisse l'importazione del file ed il suo corretto salvataggio nel database.

#### Importazione delle informazioni statiche - Codifica

A livello *frontend*, si è realizzata la finestra di dialogo descritta in precedenza. Il risultato è mostrato in Figura 3.10.
A livello *backend*, per gestire l'importazione del file, è stato realizzato un nuovo model nella parte *OTA*, chiamato semplicemente *Royal*, avente i seguenti metodi:

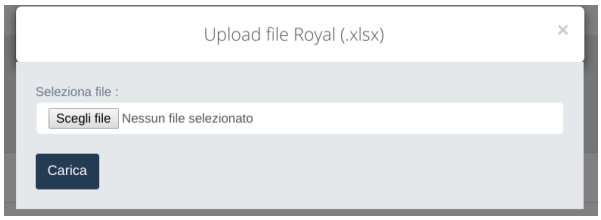


**Figura 3.10:** Schermata di caricamento del file passato da ISTINFOR

| Metodo | Descrizione |
|---|---|
| + parse(fileName: string) : boolean | Elabora il file avente nome (e percorso) 'fileName'. Restituisce true se l'elaborazione è andata a buon fine, false altrimenti . |
| - loadPorts () : boolean | Viene invocato da *parse*, si occupa di leggere i porti dal file e caricarli nella tabella *Cruises_Ports* del *DataExchange* (che poi viene sincronizzata, ad ogni integrazione, con l'omonima tabella dell'*OTA*). Restitusce *true* in caso di successo, *false* altrimenti. |
| - loadShips () : boolean | Viene invocato da *parse*, si occupa di leggere le navi dal file e caricarle nella tabella *Cruises_Ships* del *DataExchange* (che poi viene sincronizzata, ad ogni integrazione, con l'omonima tabella dell'*OTA*). Restitusce *true* in caso di successo, *false* altrimenti. |
| - loadDecks () : boolean | Viene invocato da *parse*, si occupa di leggere i ponti delle varie navi dal file e caricarli nella tabella *Cruises_Decks* del *DataExchange* (che poi viene sincronizzata, ad ogni integrazione, con l'omonima tabella dell'*OTA*). Restitusce *true* in caso di successo, *false* altrimenti. |
| - loadCategories () : boolean | Viene invocato da *parse*, si occupa di leggere le categorie di cabine delle varie navi dal file e caricarle nella tabella *Cruises_CabinCategories* del *DataExchange* (che poi viene sincronizzata, ad ogni integrazione, con l'omonima tabella dell'*OTA*). Restitusce *true* in caso di successo, *false* altrimenti. |

È stato poi creato un nuovo metodo *caricaRoyal* sul controller *Back* dell'*OTA*, che prende il file caricato dal *frontend* e lo passa al metodo *parse* del model *Royal* sopra descritto.

### Utilizzo di FDF - Progettazione

Nel file *.xlsx* fornito da *ISTINFOR* mancano tutte le informazioni inerenti a itinerari, prezzi e partenze delle crociere. Queste sono invece fornite dal WebService **FDF**, che quindi si è dovuto implementare. L'interazione con tale WebService avviene tramite chiamate SOAP ad appositi URL forniti da *ISTINFOR*.
Essendoci stati forniti solo degli URL di sviluppo, le chiamate restituiscono dati verosimili ma non veri. Inoltre, *ISTINFOR* utilizza un firewall: vengono accettate richieste provenienti solo da IP noti. Siccome il server di sviluppo utilizzato era all'interno della rete dell'ufficio, con IP dinamico e non esposto, è stato necessario

creare un ambiente online di test, da affiancare a quello di produzione (all'indirizzo [6] e [7]).
Premesso ciò, la struttura di una chiamata SOAP accettata da *FDF* è la seguente:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
    soap/envelope/" xmlns:ns1="http://tempuri.org/WebService/
    Service1">
  <SOAP-ENV:Body>
    <ns1:PROCEDURA>
      <ns1:MessageXML>
        <PROCEDURA>
          <Header>
            <CruiseLineCode>RCCL</CruiseLineCode>
            <SubsystemId>2</SubsystemId>
            <AgencyId1>041178383</AgencyId1>
            <AgencyId2>041178383</AgencyId2>
            <Currency>EUR</Currency>
            <AgencyConsumer>A</AgencyConsumer>
          </Header>
          <PROCEDURA>
            <PARAMETRO/>
          </PROCEDURA>
        </PROCEDURA>
      </ns1:MessageXML>
    </ns1:PROCEDURA>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il tag *Header* ed il suo contenuto sono comuni a tutte le richieste. Bisogna poi sostituire il nome *PROCEDURA* con quello della funzione che si vuole invocare (ad esempio *SearchBySea*) e, in base a quanto definito nella documentazione, includere una lista di parametri (allo stesso livello del tag *Parametro*). Prima di invocare una procedura generica, però, è necessario fare una chiamata alla funzione di login, che permette di autenticarsi, come mostrato in Figura 3.11.

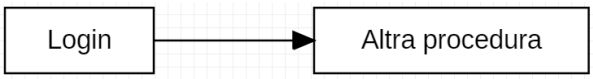


**Figura 3.11:** Flow chart dell'invocazione di una qualsiasi procedura FDF.

### Utilizzo di FDF - Codifica

Tutte le chiamate ai WebService vengono gestite dal *DataExchange*, all'interno del quale sono state create due nuove classi per semplificarne l'interazione: *Royal_Model* e *Royal_WS*.
La prima è un model che serve ad effettuare le chiamate ai WebService forniti, e si compone dei seguenti metodi:

| Metodo | Descrizione |
|---|---|
| - buildHeader(lineCode: string) : void | Costruisce e salva in una variabile interna alla classe l'intestazione della richiesta, popolando contenuto il tag *<CruiseLineCode>* con il valore del parametro *lineCode* passato. |
| - callWS (xml: string, function: string) : SimpleXMLObject | Fa una richiesta al WebService, chiamando la procedura remota *function*, utilizzando come *payload* il valore del parametro *xml*. Restituisce la risposta del WebService come istanza di SimpleXMLObject (che permette di muoversi agevolmente tra i vari tag e attributi), oppure NULL in caso si verifichi un errore di parsing (ad esempio la risposta non sia in XML ben formattato). |
| + login (lineCode: string) : boolean | Effettua la chiamata alla procedura remota di login, avvalendosi della funzione *buildHeader*, alla quale viene passato il parametro *lineCode*. |
| + requestPricesSea (sea: string, date: string, guests: int) : SimpleXMLObject | Effettua una RPC al metodo *RequestSearchBySeaPricing* del WebService, con i parametri *Sea*, *Date* e *Guests* valorizzati in base a quanto passato. |

Il metodo *RequestSearchBySeaPricing* di *FDF* permette di effettuare una ricerca del miglior prezzo per ogni categoria di cabina di tutte le partenze di tutte le navi gestite dal sistema (quindi Royal, Celebrity e Azamara). Nella ricerca possono essere inserite delle restrizioni sul numero di passeggeri, sull'area geografica (mare) di navigazione e su un intervallo di date (ad esempio crociere che partono dal 3 settembre 2018 al 10 ottobre 2019).
La chiamata a *RequestSearchBySeaPricing* (quindi al metodo *requestPricesSea* di *Royal_Model*) viene utilizzata dal meccanismo schedulato di integrazione per popolare le informazioni inerenti agli itinerari disponibili, alle date di partenza di ogni itinerario e ai prezzi di ogni categoria di cabina di ogni data di partenza di ogni itinerario. Tale elaborazione richiede molto tempo (fino a 40 minuti), per tanto viene eseguita durante la notte (è stato deciso di schedularla alle 2 del mattino, dato che non c'é un orario preciso di rilascio dei nuovi aggiornamenti).
Per permettere l'invocazione di tale metodo è stato realizzato un controller apposito, chiamato *Royal_WS*, il cui contenuto è il seguente:

| Metodo | Descrizione |
|---|---|
| + load_PricesSea() : void | Tramite la chiamata al metodo *requestPrices-Sea* del model *Royal_ Model*, sincronizza, per ogni itinerario, la lista delle partenze (incluse disponibilità di cabine, prezzi e in generale tutto quanto restituito dalla funzione) con il database del *DataExchange*. Grazie al meccanismo dei *magic method* descritto in sezione 3.2.5, esso può essere invocato direttamente via URL. |

### Utilizzo di RCCL - Progettazione

Il metodo *RequestSearchBySeaPricing* di *FDF*, ai fini dell'integrazione dati, presentava un grande difetto: non restituiva il dettaglio dell'itinerario, ma vi si riferiva utilizzandone solo il codice. Ciò rappresentava un problema sia ai fini della ricerca offerta dal *Booking Engine* che della visualizzazione dei dettagli di una crociera. In tale pagina, infatti, è presente la lista di destinazioni toccate dalla crociera, mostrata in Figura 3.12. Per reperire queste informazioni si sarebbe potuto mappare a mano ciascun itinerario in base alle informazioni contenute nei cataloghi cartacei delle tre compagnie, ma sarebbe stato un lavoro enorme (gli itinerari sono circa 1400 in totale) che si sarebbe dovuto ripetere costantemente nel tempo, in quanto essi cambiano ogni manciata di mesi.


Itinerario Crociera

| Giorno | Porto di partenza | Arrivo | Partenza |
|---|---|---|---|
| 1 | AMBURGO, GERMANIA | 08:00 | 20:00 |
| 2 | NAVIGAZIONE | -: | -: |
| 3 | OSLO, NORVEGIA | 07:30 | 15:00 |
| 4 | NAVIGAZIONE | -: | -: |
| 5 | NYNASHAMN, SWEDEN | 10:00 | 18:30 |
| 6 | TALLINN, ESTONIA | 07:00 | 17:00 |
| 7 | SAN PIETROBURGO, RUSSIA | 00:01 | 23:59 |
| 8 | SAN PIETROBURGO, RUSSIA | 07:00 | 18:00 |
| 9 | HELSINKI, FINLANDIA | 06:00 | 14:00 |
| 10 | AMBURGO, GERMANIA | 06:00 | -: |

**Figura 3.12:** Dettaglio dell'itinerario visualizzato dal *frontend*.

Per evitare tanto lavoro si è deciso di implementare, all'interno del metodo *load_ PricesSea* della classe *Royal_ WS*, delle RPC a *RequestItinerary*, fornito dal WebService *RCCL*, che restituisce le informazioni dettagliate dell'itinerario corrispondente ai parametri passati (codice della nave, data di partenza e codice itinerario). Dato che il protocollo di comunicazione è analogo a quello utilizzato per *FDF*, si è deciso (per questo e per

tutti le altre RPC utilizzate di *RCCL*) di riutilizzare la classe *Royal_ Model*, andando a definire nuovi metodi in base alle necessità.

### Utilizzo di RCCL - Codifica

In questo caso è stato aggiunto il seguente metodo:

| Metodo | Descrizione |
|---|---|
| + requestItinerary(ship: string, date: string, itinerary: string) : SimpleXMLObject | Effettua una RPC a *RequestItinerary* del Web-Service, con i parametri *Ship*, *Date* e *Itinerary* valorizzati in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |

Le informazioni ritornate da questo metodo includono anche la lista di destinazioni dell'itinerario, ma la lingua in cui i nomi di tali destinazioni (porti, città, stati) vengono restituiti è mista. Ad esempio, per riferirsi alle isole delle Azzorre, viene usato tanto il nome italiano "Azzorre" quanto il nome spagnolo "Azores". La stessa cosa vale per Danimarca, Scozia, Australia, Emirati Arabi, Canarie, Belgio ecc. È stato dunque necessario realizzare un mapping manuale dei nomi, che permettesse di tradurre i nomi in lingua italiana.

Per quanto riguarda l'integrazione nel flusso di prenotazione delle tariffe *FIBOS*, analogamente al lavoro svolto per i *vuoto per pieno*, sono stati creati ed implementati nuovi metodi nel controller *WS_ Cruises* che utilizzano nuovi metodi creati nel model *Royal_ Model*. Sono state utilizzate le seguenti procedure di *RCCL*:

| Procedura | Descrizione |
|---|---|
| RequestCategories | Dati in input nave (Ship), data di partenza (Date), numero ed età dei passeggeri (Guests), restituisce una lista di tariffe disponibili per quella partenza, complete di prezzi per ogni categoria di cabina, tasse portuali ed eventualmente oneri aggiuntivi. |
| RequestCabins | Dati in input nave (Ship), data di partenza (Date), numero ed età dei passeggeri (Guests) e codice della tariffa (FareCode), restituisce una lista di cabine disponibili che soddisfano i criteri definiti dai parametri passati. |

| | |
|---|---|
| RequestPricing | Dati in input nave (Ship), data di partenza (Date), numero ed età dei passeggeri (Guests), codice della tariffa (FareCode) e della cabina (Cabin), restituisce una quotazione veritiera che soddisfa i criteri definiti dai parametri passati. |
| RequestBooking | Dati in input nave (Ship), data di partenza (Date), numero, età e dati anagrafici dei passeggeri (Guests), codice della tariffa (FareCode) e della cabina (Cabin) e la tipologia di prenotazione che si vuole effettuare (AgreementSt, da porre a OF in caso si voglia fare un'opzione, mentre a BK in caso di prenotazione vera e propria), restituisce il codice di prenotazione (BookingId), assieme all'ammontare di denaro e relative scadenze dei pagamenti. |
| RequestRetrieve | Dato in input il numero di prenotazione (BookingId), restituisce le informazioni dettagliate (sia inerenti all'itinerario, sia allo stato dei pagamenti) della prenotazione corrispondente. |

Per effettuare le RPC sopra descritte, sono stati realizzati i seguenti nuovi metodi all'interno della classe *Royal_Model*:

| **Metodo** | **Descrizione** |
|---|---|
| + categoryAvailability(ship: string, date: string, guests: array) : SimpleXMLObject | Effettua una RPC a *RequestCategories* del WebService, con i parametri *Ship*, *Date* e *Guests* valorizzati in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |
| + cabinAvailability(ship: string, date: string, guests: array, fareCode: string) : SimpleXMLObject | Effettua una RPC a *RequestCabins* del WebService, con i parametri *Ship*, *Date*, *Guests*, *FareCode* valorizzati in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |

| | |
|---|---|
| + pricingInformation(ship: string, date: string, cabin: string, fareCode: string, guest: array) : SimpleXMLObject | Effettua una RPC a *RequestPricing* del WebService, con i parametri *Ship*, *Date*, *Guests*, *FareCode*, *Cabin* valorizzati in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |
| + book(ship: string, date: string, cabin: string, fareCode: string, guest: array, agreementSt: string) : SimpleXMLObject | Effettua una RPC a *RequestBooking* del WebService, con i parametri *Ship*, *Date*, *Guests*, *FareCode*, *Cabin*, *AgreementSt* valorizzati in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |
| + retrieveBooking(bookingNumber: string) : SimpleXMLObject | Effettua una RPC a *RequestRetrieve* del WebService, con il parametro *BookingId* valorizzato in base a quanto passato. Ritorna la risposta in forma di oggetto SimpleXMLObject, o NULL in caso di risposta mal formattata. |

Infine, per invocare ed elaborare i risultati dei nuovi metodi scritti in *Royal_ Model*, sono state aggiunte le seguenti funzioni a *WS_ Cruises*:

| **Metodo** | **Descrizione** |
|---|---|
| + Fibos_categoryAvailability (supplier: int, cruiseID: string, guest: array) : array | Restituisce un array contenente la lista delle categorie di cabina effettivamente disponibili, interrogando il WebService tramite l'invocazione del metodo *categoryAvailability* di *Royal_ Model*. |
| + Fibos_cabinAvailability (supplier: int, cruiseID: string, guest: array, fareCode: string) : array | Restituisce un array contenente la lista delle singole cabina effettivamente disponibili, interrogando il WebService tramite l'invocazione del metodo *categoryAvailability* di *Royal_ Model*, tenendo conto dei parametri passati (quindi anche del fareCode, ovvero della tariffa scelta allo step precedente). |
| + Fibos_requestPricing (supplier: int, cruiseID: string, guest: array, fareCode: string, cabin: string) : array | Restituisce un array contenente una quotazione veritiera per la cabina selezionata (tenendo conto di tariffa, crociera, cabina, selezionate e del numero ed età dei passeggeri), interrogando il WebService tramite l'invocazione del metodo *pricingInformation* di *Royal_ Model*. |

| | |
|---|---|
| + Fibos_requestBooking (supplier: int, cruiseID: string, guest: array, fareCode: string, cabin: string, agreementSt: string) : array | Effettua una prenotazione in base ai parametri passati (tariffa, crociera, cabina, numero ed età dei passeggeri, tipo di prenotazione), interrogando il WebService tramite l'invocazione del metodo *book* di *Royal_Model*. |
| + Fibos_requestBookingInformation (bookingNumber: string) : array | Restituisce le informazioni inerenti alla prenotazione passata come parametro, interrogando il WebService tramite l'invocazione del metodo *retrieveBooking* di *Royal_Model*. |

#### Problemi riscontrati

Nell'implementazione dei due WebServices sopra menzionati, sono stati riscontrati dei problemi, principalmente con la documentazione fornitaci da *ISTINFOR*. All'inizio, infatti, ci sono stati forniti i manuali aggiornati a febbraio 2014, e molti parametri delle procedure da noi usate non corrispondevano a quanto scritto nel manuale. Dopo numerose sollecitazioni, siamo riusciti ad ottenere la guida aggiornata all'ultima release (febbraio 2018).
Un altro problema riscontrato, forse quello che ci ha fatto perdere più tempo, è stato l'eterogeneità dei formati delle date richiesti dai parametri/tag/attributi delle varie procedure. Infatti, alcune di esse richiedevano che la data fosse espressa in formato YYYY-mm-dd (ovvero 2018-09-27), altre in formato YYYYmmdd (20180927), altre ancora in formato dd/mm/YYYYY (27/09/2018) o addirittura mm/dd/YYYY (09/27/2018). Al di là delle numerose conversioni necessarie per adattarle al formato presente nel database (che è YYYY-mm-dd), il problema è stato che il manuale si è rivelato ambiguo riguardo al formato corretto da usare (o non veniva specificato o, addirittura, veniva specificato un formato sbagliato). Visto anche l'elevato tempo di risposta (settimane) del supporto tecnico, soprattutto probabilmente perché lo stage si è svolto durante il periodo estivo tipico delle ferie, abbiamo dovuto procedere molto spesso per tentativi, perdendo molto tempo.

### 3.6.3 Test

Anche in questo caso, vista l'assenza di una suite di testing, si sono svolte delle verifiche a mano di congruenza tra i dati restituiti da i WebService e quelli presenti nel database. Inoltre, grazie alle credenziali di accesso forniteci da *Primarete*, è stato possibile verificare la congruenza tra quanto presente nel database e quanto visualizzato dal portale di prenotazione ufficiale del gruppo Royal/Celebrity/Azamara. I dati in nostro possesso, sebbene derivanti da canali di test e non di produzione, erano in pratica dati reali, solo semplicemente non aggiornati.

# Capitolo 4

# Conclusioni

## 4.1 Raggiungimento degli obiettivi

La pianificazione in termine di ore totali è stata perfettamente rispettata: lo stage si è svolto in 310 ore, rispettando le scadenze settimanali pianificate nella sezione 2.4.2. Ciò, purtroppo, significa che non c'è stato spazio per lo svolgimento delle attività **opzionali** descritte sezione 2.2.5.
Riassumendo, lo stato di soddisfacimento degli obiettivi è stato dunque il seguente:

| | **Obbligatori** | Soddisfatto |
|---|---|---|
| Ob1 | Interazione con il database SQL Server attraverso le librerie del framework Codeigniter | Si |
| Ob2 | Realizzazione integrazione flat-file di un nuovo fornitore con il Data Exchange del *Booking Engine* | Si |
| Ob3 | Aggiunta prodotti e tariffe del nuovo fornitore ai risultati della ricerca lato *frontend* del *Booking Engine* | Si |
| Ob4 | Esecuzione test e redazione documentazione sul lavoro svolto | Si |
| | **Desiderabili** | Soddisfatto |
| D1 | Realizzazione del registro carichi/scarichi tariffe "vuoto per pieno" come funzionalità lato *backend* del *Booking Engine* | Si |
| D2 | Interrogazione web-service in tempo reale per sincronizzare prezzi e disponibilità del nuovo fornitore con il Data Exchange | Si |
| D3 | Realizzazione conferma prenotazione al fornitore come funzionalità lato *frontend* del *Booking Engine* | Si |
| | **Opzionali** | Soddisfatto |
| Op1 | Analisi e realizzazione di nuove funzionalità | No |

Avendo raggiunto tutti gli obiettivi obbligatori e desiderabili prefissati, considero l'esito dello stage più che soddisfacente, anche perché buona parte del codice da me sviluppato è stato portato direttamente in produzione.

## 4.2 Competenze acquisite

### 4.2.1 Competenze tecnologiche

A livello tecnologico ho avuto modo di ampliare ed approfondire le mie conoscenze, data anche la mia esperienza pregressa in questo campo.
Ho avuto modo di interagire con il framework *Codeigniter*, che devo dire ho imparato ad apprezzare molto, soprattutto per la sua semplicità. Posso sicuramente affermare che riutilizzerò questo framework nei prossimi progetti in PHP che andrò a realizzare.
Ho avuto anche l'occasione di interfacciarmi con *Microsoft SQL Server* che però, in fin dei conti, non differisce poi più di tanto rispetto agli altri DBMS da me già utilizzati in passato.

### 4.2.2 Competenze metodologiche

A livello metodologico, invece, posso affermare di aver ricevuto un grande valore aggiunto da questa esperienza. Ho potuto "toccare con mano" cosa significhi lavorare con metodo, ed i vantaggi che ciò porta a livello di produttività. Infatti, avere un metodo di lavoro permette di realizzare prodotti di qualità più elevata, che rispondono in modo più affine alle esigenze del committente. Ho anche potuto riscontrare come il modello evolutivo descritto in sezione 1.3.1 permetta di rispondere molto bene alle esigenze mutevoli del mercato (anzi, sarebbe meglio dire del committente tipicamente indeciso). Posso dunque affermare di aver acquisito una *way-of-working* che sicuramente andrò ad applicare ai miei progetti "amatoriali" che sto attualmente portando avanti.

## 4.3 Valutazione sul rapporto azienda-università

Ho iniziato a programmare da completo autodidatta. La scuola superiore che ho frequentato ha contribuito a darmi un'infarinatura accademica riguardo il mondo dell'informatica, ma sono sempre stato un cosiddetto "smanettone". Per imparare qualcosa, ho sempre prediletto l'aspetto pratico a quello teorico, perché permette di ricevere subito un risultato tangibile del proprio lavoro. All'inizio del percorso della laurea triennale, devo dire di essere rimasto un po' deluso dall'eccessivo (a mio parere dell'epoca) approccio teorico nello studio di questa materia. Con l'avanzare del tempo, però, ho imparato che è utile capire quello che si sta facendo, e per capirlo è necessario valutare anche l'approccio teorico al problema che si sta cercando di risolvere.
A posteriori, ora che, con la discussione di questo documento, ho concluso il mio percorso di laurea triennale, penso che l'approccio teorico offerto da questo corso di laurea possa giovare a tanti "smanettoni" come me, anche perché il mondo dell'informatica si evolve ad una velocità molto elevata, che rende molto difficile starne al passo. Ritengo tuttavia sarebbe una buona idea avviare collaborazioni, nel limite del possibile, con "entità esterne all'aula" (aziende, istituzioni) per lo svolgimento dei progetti tipici di alcuni corsi (ad esempio Tecnologie Web). Così facendo, probabilmente, si avrebbero degli studenti più abili nella risoluzione di problemi "reali", preparati ad affrontare

il mondo del lavoro e sarebbe data loro la possibilità di esplorare tecnologie magari innovative ma sempre affini al corso (ad esempio Node.js), un po' come accade già con Ingegneria del Software. Probabilmente, infine, sarebbe anche più motivante per gli studenti stessi, in quanto realizzerebbero qualcosa di non fine a se stesso.

# Glossario

**AJAX** Acronimo di Asyncronous Javascript And XML, tecnica che prevede lo scambio di dati in background tra server e browser, utilizzando richieste HTTP asincrone [20]. 17, 19, 33, 49

**API** acronimo di Application Programming Interface. Serie di convenzioni adottate dagli sviluppatori di software per definire il modo con il quale va richiamata una determinata funzione di un'applicazione. L'impiego di API comuni ha lo scopo di rendere più omogenea l'interfaccia e di facilitare l'interazione di programmi che diversamente risulterebbero molto differenti e distanti fra loro [9]. 12, 20, 21, 49

**DBMS** Data Base Management System, sistema software progettato per la gestione (creazione, manipolazione, interrogazione) di basi di dati (database) [15]. iii, 3, 12, 15, 16, 46, 49

**DOM** Acronimo di Document Object Model, rappresentazione in forma di albero di oggetti del contenuto della pagina HTML (Document) a cui si riferisce [18]. 49

**Framework** Architettura software che include degli strumenti (classi, metodi) con lo scopo di semplificare lo sviluppo, facilitando così il lavoro del programmatore [17]. 12, 14, 17, 20, 45, 46, 49

**IDE** Un ambiente di sviluppo integrato (in lingua inglese integrated development environment ovvero IDE, anche integrated design environment o integrated debugging environment, rispettivamente ambiente integrato di progettazione e ambiente integrato di debugging), in informatica, è un software che, in fase di programmazione, aiuta i programmatori nello sviluppo del codice sorgente di un programma.

Spesso l'IDE aiuta lo sviluppatore segnalando errori di sintassi del codice direttamente in fase di scrittura, oltre a tutta una serie di strumenti e funzionalità di supporto alla fase di sviluppo e debugging [8]. 9, 49

**Incremento** Procedere per aggiunta ad una base. 5, 49

**Iterazione** Procedere per rivisitazioni (può includere un incremento o addirittura un decremento).
L'iterazione è un processo di durata non determinabile (anche potenzialmente infinita). 5, 49

**jQuery** Una tra le più diffuse librerie Javascript, che agevola la manipolazione del DOM.. 17, 49

**JSON** Acronimo di JavaScript Object Notation, formato utilizzato per la rappresentazione di oggetti sotto forma di stringa [19]. 17, 19, 31, 50

**MVC** Acronimo di *Model View Controller*, è un design pattern architetturale in grado di separare la logica di presentazione dalla logica di business. Si compone di tre tipologie di componenti (classi): Modelli, che rappresentano i dati processati dall'applicazione, Viste che rappresentano l'interfaccia grafica dell'applicazione e Controller, che accetta in input il modello e lo converte in comandi per la vista [12]. ix, 16, 21, 50

**RDBMS** Relational Data Base Management System, DBMS basato sul modello relazionale [16]. 3, 50

**RPC** Remote Procedure Call, chiamata di una procedura remota [13]. 39–43, 50

**SEO** Acronimo di *Search Engine Optimizazion*, definisce tutte le attività per migliorare il posizionamento di un determinato sito web nei motori di ricerca [11]. 14, 50

**SOAP** SOAP è un protocollo per lo scambio di messaggi tra componenti software, che permette di chiamare procedure remote (RPC Call, Remote Procedure Call). Richieste e risposte SOAP sono codificate con XML [14]. 37, 38, 50

**Tariffa** Gruppo di prezzi di una cabina, accomunati da uno o più fattori. La stessa cabina può avere due tariffe diverse (a prezzi diversi), perchè magari la prima include dei servizi che la seconda non ha (come bibite illimitate). 31, 50

**Tempo di risposta** Tempo impiegato dal server per elaborare l'output, che verrà poi scaricato dal client. Il tempo di risposta, quindi, non include il tempo di download dell'output. 23, 24, 50

**WebService** Un WebService é un sistema software progettato per supportare interazioni macchina-macchina su una rete. Dispone di un'interfaccia descritta da un linguaggio processabile da una macchina (nello specifico WSDL). Altri sistemi interagiscono con il Web service in una maniera definita in base alla sua descrizione usando messaggi SOAP, tipicamente convogliati usando HTTP con serializzazione XML assieme ad altri standard Web [10]. 12–15, 18, 20, 33–44, 50

**WISP** Acronimo di Windows (Server) - IIS - SQL Server - PHP. Viene utilizzato da WebPD per indicare lo stack tecnologico utilizzato dal *Booking Engine.*. 17, 50

# Bibliografia


[1] *Prezzi di SQL Server - consultato 13/09/2018*
https://bit.ly/2rECrAS

[2] *Studio comparativo sulle performance di SQL Server PostgreSQL Oracle XE e MySQL - consultato 13/09/2018*
https://bit.ly/2MtTE8e

[3] *Differences Between Git and SVN - consultato 13/09/2018*
https://bit.ly/2p4NzWI

[4] *URL del sito CrociereRegalo (parte OTA)*
https://www.crociereregalo.it

[5] *URL del sito CrociereRegalo (parte DataExchange)*
https://data.crociereregalo.it

[6] *URL del sito di sviluppo CrociereRegalo (parte OTA)*
http://primaretetest.webpd.it/crociereregalo

[7] *URL del sito di sviluppo CrociereRegalo (parte DataExchange)*
http://primaretetest.webpd.it/dataExchange

[8] *Integrated development environment - consultato 18/09/2018*
https://bit.ly/2p4NzWI

[9] *Cosa sono le API e a cosa servono - consultato 18/09/2018*
https://bit.ly/2xm5qgq

[10] *Web Services Architecture - consultato 18/09/2018*
https://www.w3.org/TR/ws-arch/#introduction

[11] *Guida SEO all'Ottimizzazione per Motori di Ricerca - consultato 18/09/2018*
https://bit.ly/2QGWt9x

[12] *Advanced ActionScript 3 with Design Patterns*
Joe Lott, Danny Patterson
*Pagina 46*

[13] *Remote Procedure Call - consultato 18/09/2018*
https://bit.ly/2rjaUVo

[14] *Remote Procedure Call - consultato 18/09/2018*
https://www.w3schools.com/xml/xml_soap.asp

[15] *Data Base Management System - consultato 18/09/2018*
https://bit.ly/2HECZMV

[16] *RDBMS - consultato 18/09/2018*
https://bit.ly/1MLo1Q4

[17] *Framework - consultato 18/09/2018*
https://bit.ly/2QGe2Gr

[18] *JavaScript HTML DOM - consultato 18/09/2018*
https://www.w3schools.com/js/js_htmldom.asp

[19] *Introducing JSON - consultato 18/09/2018*
https://www.json.org/

[20] *AJAX Introduction - consultato 18/09/2018*
https://www.w3schools.com/xml/ajax_intro.asp