Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Tesi di Laurea in Ingegneria Informatica e Automatica

Progettazione e realizzazione di una piattaforma per generare e distribuire contenuti editoriali in ambito mobile

Relatore interno:
Prof. Gabriele Di Stefano

Laureando:
Luca Di Stefano

Relatore esterno:
Dott. Alessandro Zavattaro

Anno Accademico 2012-2013

Indice

1 Introduzione **5**

2 Obiettivi **8**

3 Progettazione **13**
3.1 Diagramma dei componenti 13
3.2 Integrazione con CMS preesistenti 13
3.3 Sistema di produzione 15
3.3.1 Front-end 16
3.3.2 Back-end 16
3.4 Libreria Data Transfer Object 19
3.5 Web service 20
3.6 Applicazioni mobile 21

4 Tecnologie utilizzate **24**
4.1 CMS: WordPress 24
4.2 Visualizzazione *responsive*: Treesaver.js 25
4.3 Framework .NET e Common Language Infrastructure 26
4.3.1 Compilazione: Common Intermediate Language 26
4.4 Il linguaggio C# 27
4.4.1 Caratteristiche del linguaggio 27
4.5 Mono e Xamarin 31
4.5.1 Il progetto Mono 31
4.5.2 Architettura di un'applicazione Xamarin 32
4.6 Altre tecnologie 32
4.6.1 ADO.NET Entity Framework 32
4.6.2 Windows Communication Foundation 33

4.6.3 OAuth 34

5 Implementazione **35**
5.1 CMS: plugin di integrazione per WordPress 35
5.2 Base di dati 37
5.3 Sistema di produzione 37
5.3.1 Libreria Data Transfer Object 37
5.3.2 Back-end 38
5.3.3 Front-end 40
5.4 Web service 42
5.4.1 Microsoft Web API 42
5.4.2 Handler HTTP 44
5.5 Applicazioni Mobile 45
5.5.1 Libreria cross-platform `GutenbergCore` 46
5.5.2 Interfaccia Android 48
5.5.3 Windows Phone 51

6 Conclusioni e sviluppi futuri **55**

Introduzione

Il progetto descritto in questa tesi è frutto del mio tirocinio formativo presso l'azienda **Aubay Research & Technologies S.p.A.**, una società di consulenza e *system integration* attiva dal 1997.

Il tirocinio si è svolto nei mesi di gennaio e febbraio 2014 negli uffici dell'azienda situati a Carsoli, per una durata complessiva di 280 ore.

La sede di Carsoli è strettamente legata a clienti di dimensioni medio-grandi che operano nel settore dell'editoria e dei *media*, ai quali fornisce soluzioni e servizi IT riguardanti:

- Ottimizzazione dei processi;
- Sviluppo di infrastrutture tecnologiche ed applicazioni;
- Integrazione con sistemi *legacy*.

Un esempio di quanto appena esposto è la gestione dell'intero ciclo di vita delle architetture software che supportano i prodotti Banche Dati Professionali del gruppo 24 Ore [13].

L'azienda sta attualmente valutando la fattibilità di soluzioni atte a facilitare l'integrazione di un modello di distribuzione digitale orientato a dispositivi *mobile* (smartphone e tablet) nell'offerta dei propri clienti.

Un utente dovrà quindi essere in grado di acquistare, mediante un'applicazione installata sul proprio dispositivo, una versione digitale delle riviste prodotte dal cliente: il cliente, d'altro canto, potrà realizzare e distribuire riviste differenti per genere e prezzo.

La necessità di sviluppare un simile prodotto deriva dalla sempre crescente diffusione di questi dispositivi all'interno del mercato italiano: l'analisi *EMEA Smart Connected Device Tracker* curata dalla società IDC registra

ad esempio, per il terzo trimestre del 2013, un aumento anno su anno del 40,9% nella vendita di tablet e del 33,8% per quanto riguarda il mercato degli smartphone [8].

Naturale effetto di questa tendenza è una netta evoluzione della rete Internet *mobile*, i cui contorni sono stati tracciati da uno studio a cura della compagnia Cisco [2]. Lo studio stima una crescita del traffico mobile sulla rete globale dai circa 1,5 EB/mese[1] del 2013 a quasi 16 EB/mese nel 2018, con un tasso di crescita annuo del 61%; misura inoltre una crescita del traffico mobile pari all'81% nel corso del 2013.

Analisi sul territorio italiano citate dallo stesso studio riportano un aumento del 34% del traffico mobile in Italia [1].

Nello studio della soluzione abbiamo tenuto conto di una serie di obiettivi e restrizioni legate alla realtà in cui operano l'azienda e i suoi clienti:

1. La soluzione dovrà gestire l'intero ciclo di vita del prodotto, dalla produzione alla diffusione tra gli utenti finali.

2. I clienti avranno bisogno di una piattaforma facilmente integrabile con il software attualmente in uso nella gestione dei propri contenuti editoriali.

   Una soluzione che costringesse le redazioni ad adottare un nuovo *workflow* interno sarebbe eccessivamente onerosa per il cliente in termini di costo di *deployment* e formazione del personale, e in generale sarebbe di difficile introduzione sul mercato.

3. La soluzione dovrà essere facilmente *scalabile* in base alle esigenze del cliente.

4. Il prodotto dovrà mantenere un'elevata *fault-tolerance*, specialmente per quanto riguarda le funzionalità direttamente raggiungibili dall'utente finale.

5. L'applicazione *mobile* dovrà garantire un'esperienza utente simile a quella di una rivista cartacea, ma senza inficiare sull'usabilità generale: ad esempio, l'impaginazione dovrà essere dinamica e basata sulle dimensioni dello schermo.

6. L'applicazione dovrà essere compatibile con una buona percentuale dei dispositivi attualmente sul mercato.

[1] Exabyte al mese.

7. L'azienda dovrà studiare pratiche di sviluppo *mobile* tali da richiedere un team di sviluppo e manutenzione ragionevolmente ridotto. L'idea di realizzare un'applicazione nativa per ciascun sistema operativo è quindi da scartare, a causa dell'estrema varietà di competenze richieste e dei conseguenti costi per l'azienda ed il cliente: gli sviluppatori dovrebbero programmare in linguaggi differenti (Java, C#, Objective-C) e gestire tre progetti concettualmente interconnessi, ma di fatto separati tra loro.

   Una soluzione può essere costituita da uno dei molti framework di sviluppo multipiattaforma disponibili sul mercato: nello studiare pregi e difetti di ciascuno, terremo conto dell'elevata competenza della sede di Carsoli in ambito di tecnologie Microsoft (linguaggio C#, framework .NET).

Vedremo come la scelta di tecnologie particolarmente indicate in ambito *enterprise*, unita ad un'attenta progettazione iniziale, ci abbiano permesso di sviluppare una piattaforma sperimentale che, pur non essendo pronta all'uso in ambiti produttivi, soddisfacesse al meglio ciascuno di questi requisiti.

Capitolo 2
Obiettivi

La piattaforma, che indicheremo con il nome *Gutenberg*, avrà quattro componenti principali:

- Un CMS (Content Management System) per l'inserimento e la gestione degli articoli: si suppone che una redazione utilizzi già un software di questo tipo come parte del proprio *workflow*, quindi ci limiteremo a progettare un sistema di integrazione con il resto della piattaforma;
- Un sistema di produzione per generare riviste a partire da un insieme di articoli prelevati dal CMS;
- Un web service che esponga una serie di azioni (visualizzazione di una rivista, consultazione dell'elenco delle riviste acquistate, acquisto di una nuova rivista, ecc.) agli utenti autenticati;
- Un'applicazione per dispositivi *mobile* che faccia da tramite tra l'utente e il web service di cui sopra.

L'accesso alle prime due componenti sarà limitato alla rete aziendale, mentre il web service (per ovvi motivi) sarà esposto in Internet. Prima di descrivere le funzionalità di ciascuna di esse, è necessario fare chiarezza su alcuni termini che appariranno con frequenza nel corso di questa tesi, e che nella loro accezione comune potrebbero generare confusione.

- Con **rivista** indicheremo una singola uscita appartenente ad una determinata *testata*. Nel progetto si è usato il termine inglese *issue*.
- Il termine **testata** (*magazine*) indica, al contrario, una collezione di *riviste*.

Ogni testata avrà un identificativo numerico, mentre le riviste saranno individuate dal proprio “numero” unito all’id della testata di appartenenza.

Ciascuna testata avrà un proprio record nella base di dati ed una directory nel file system: quest’ultima conterrà una directory per ciascuna rivista appartenente a quel *magazine*.

La directory di una rivista conterrà:

- Un file `.html` per ciascun articolo;
- Un file `toc.json` contenente la *table of contents*, cioè l’indice degli articoli;
- Un file `index.html` contenente l’immagine di copertina;
- Le immagini incluse negli articoli e nella prima pagina.

I nomi delle directory corrispondono agli id dei relativi record nella base di dati: la directory `/1/2/` contiene quindi la rivista 2 della testata 1. La directory principale conterrà infine i file necessari a tutte le riviste (fogli di stile, file JavaScript, eccetera).

Gli *attori* che interagiranno con il sistema appena descritto saranno tre:

Redattori Con questo termine descriveremo tutti i ruoli connessi al workflow di creazione/revisione/approvazione degli articoli e che quindi interagiranno esclusivamente con il CMS. Il diagramma dei casi d'uso che presentiamo (figura 2.1) è volutamente semplificato: gli articoli vengono approvati per la pubblicazione dall'autore stesso, che avrà anche la facoltà di modificarli o cancellarli.

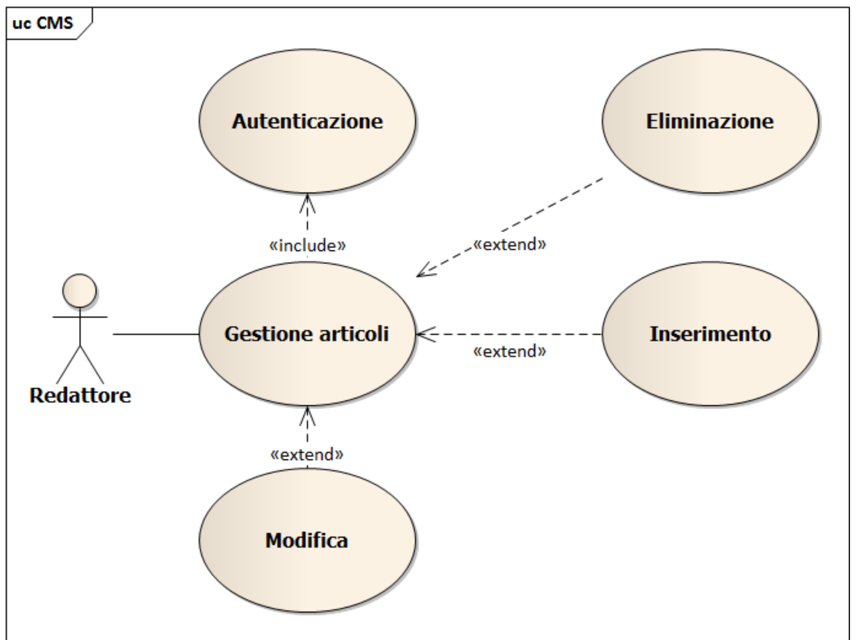


Figura 2.1: Diagramma dei casi d'uso per il CMS.

Editori L'editore ha il compito di interagire con il sistema di produzione. Ciò avverrà per mezzo di un'interfaccia web che fornirà le seguenti funzionalità (figura 2.2):

- Creazione di una nuova testata;
- Modifica dei dati relativi ad una testata esistente;
- Aggiunta di un nuovo *issue* ad una testata esistente.

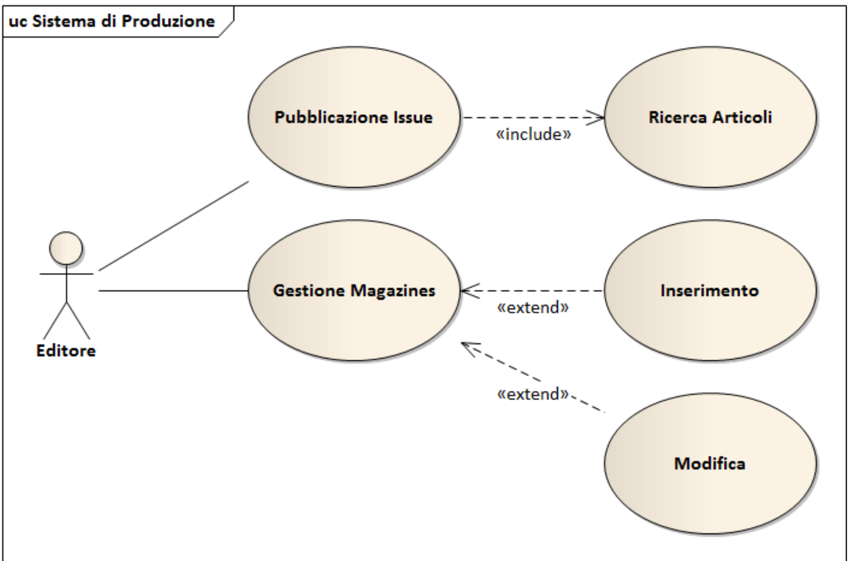


Figura 2.2: Diagramma dei casi d'uso: sistema di produzione.

Utenti Si tratta degli utilizzatori dell'applicazione mobile, che quindi interagiranno indirettamente con il web service (figura 2.3). Una volta superato un processo di autenticazione, essi avranno la possibilità di:

- Ottenere un elenco delle riviste in proprio possesso;
- Ottenere un elenco di riviste pronte all'acquisto;
- Acquistare una nuova rivista;
- Visualizzare una rivista tra quelle in proprio possesso.

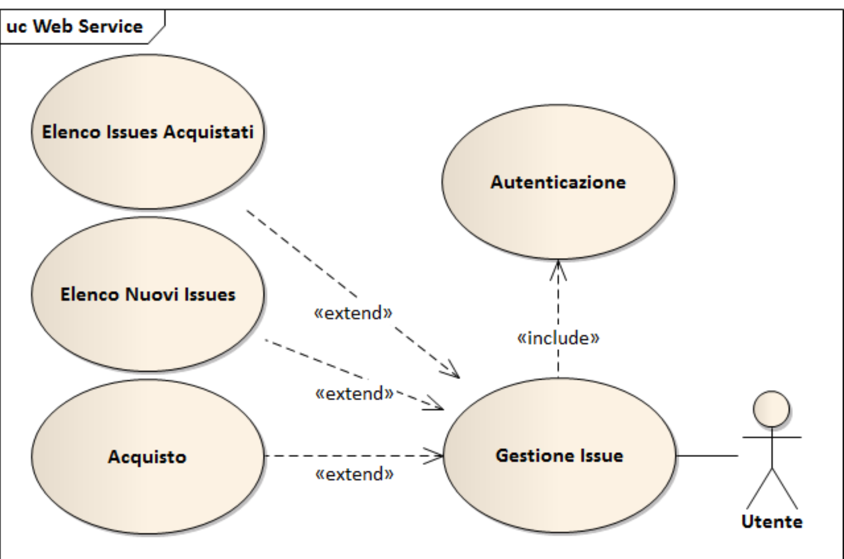


Figura 2.3: Diagramma dei casi d'uso: web service.

Progettazione

3.1 Diagramma dei componenti

Il diagramma dei componenti (figura 3.1) mostra i singoli elementi facenti parte del progetto ed il modo in cui interagiscono fra loro: ci permette quindi di avere una visione d'insieme dell'intera piattaforma, che andremo successivamente a progettare componente per componente.

3.2 Integrazione con CMS preesistenti

Abbiamo evitato di progettare e realizzare da zero un sistema di gestione contenuti (CMS): l'impegno richiesto sarebbe stato eccessivo e avremmo difficilmente ottenuto un prodotto adeguato alle necessità di un'azienda. Inoltre, come si è detto in precedenza, un prodotto integrabile in contesti di produzione già operativi avrà sicuramente un bacino di mercato più ampio.

Si è quindi giunti ad una soluzione il più possibile indipendente dal software di gestione dei contenuti: per ottenere questo risultato abbiamo comunque dovuto definire con chiarezza le modalità di comunicazione tra il sistema di produzione ed il CMS.

Per poter interagire con il resto della piattaforma, il CMS dovrà cioè mettere a disposizione un *endpoint* capace di restituire una lista di articoli in base ad una richiesta esterna.

L'endpoint verrà implementato per mezzo di un *plugin* da inserire nel CMS e risponderà a richieste di tipo `GET` aventi la seguente struttura:

```
<URL>?action=gutenberg&before=<before-date>&after=<after-date>
```

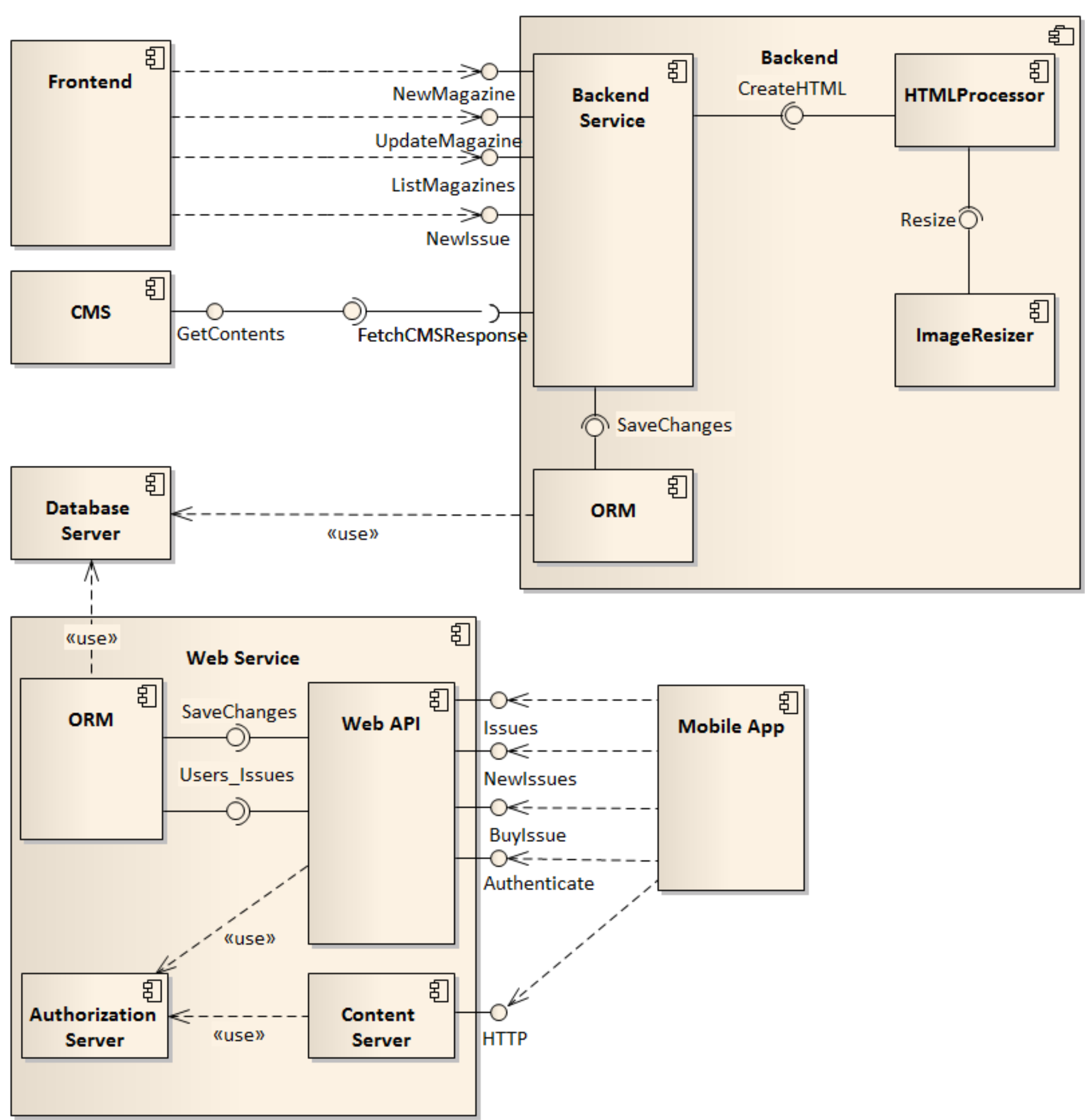


Figura 3.1: Diagramma dei componenti del progetto.

dove `<URL>` è l'indirizzo dell'endpoint e `<before-date>`, `<after-date>` sono due date in formato YYYY-MM-DD (quattro cifre per l'anno, seguite da due per il mese e due per il giorno).

La risposta conterrà un oggetto JSON [3] strutturato come segue:

```
{
    "success": (bool),
    "message": (string),
```

```
4     "before": (string),
5     "after": (string),
6     "count": (int),
7     "posts": (array)
8 }
```

`success`: `true` se la richiesta è stata accolta ed elaborata con successo.

`message`: un messaggio facoltativo (ad esempio una descrizione in caso di errore).

`before, after`: le due date incluse nella richiesta.

`count`: dimensione dell'array di articoli restituito.

L'array `posts` sarà a sua volta costituito da oggetti JSON con questa struttura:

```
1 {
2     "title": (string),
3     "author": (string),
4     "date": (string),
5     "content": (string)
6 }
```

`title`: titolo dell'articolo;

`author`: autore dell'articolo;

`date`: data di pubblicazione dell'articolo;

`content`: testo dell'articolo, in HTML.

3.3 Sistema di produzione

La progettazione della piattaforma è partita dal componente destinato a creare le riviste che verranno successivamente distribuite. Questo processo consiste nel recuperare articoli dal CMS in base alla richiesta dell'utente e nel creare per ciascuno di essi un file `.html` compatibile con il framework di visualizzazione da noi scelto.

Le immagini incluse negli articoli devono inoltre essere salvate in diverse risoluzioni, in modo che il framework possa “scegliere” quella più adeguata al dispositivo dell’utente.

Infine bisognerà creare i file `toc.json` e `index.html`, necessari alla visualizzazione della rivista stessa.

Altre funzioni che il back-end dovrà fornire comprendono la creazione o modifica di testate (*magazine*) ed il recupero dell’elenco di quelle già presenti nella base di dati.

La prima scelta di progettazione è stata di carattere architetturale: avremmo potuto costruire un sistema “monolitico” basandoci, ad esempio, sul pattern MVC e implementando la logica della piattaforma nei controller dell’applicazione. Così facendo avremmo però ottenuto un prodotto difficilmente scalabile e meno performante.

Per ovviare a queste limitazioni abbiamo separato il sistema di produzione in due elementi: un *front-end* atto a raccogliere le richieste dell’utente ed un servizio di *back-end* destinato a processare tali richieste.

3.3.1 Front-end

Non ci soffermeremo a lungo sul front-end, che seguirà i principi del pattern MVC.

Model Saranno due: *Issue* e *Magazine*, a rappresentare le entità descritte nel capitolo precedente.

View Forniranno una serie di form che consentiranno all’utente di:

- Creare/modificare un *Magazine*;
- Creare un nuovo *Issue*

in conformità con i casi d’uso descritti in precedenza.

Controller Dovranno validare i dati inseriti dall’utente per poi inviarli al back-end. La risposta proveniente da quest’ultimo verrà restituita all’utente per mezzo di una *view*.

3.3.2 Back-end

Il servizio di back-end esporrà una serie di metodi per effettuare le operazioni richieste dal front-end: tali metodi saranno descritti in un’interfaccia `IService` e implementati in una classe `Service`.

Le altre classi presenti nel diagramma 3.2 hanno le seguenti funzionalità:

`Post, Response` Classi usate per deserializzare i dati ricevuti dal CMS.

`HTMLProcessor` Verrà chiamato dalla classe `Service` per realizzare i file di cui si compone il nuovo `Issue`. Le variabili `widths` e `coverWidths` sono dei dizionari (insiemi di coppie chiave-valore) contenenti informazioni necessarie al ridimensionamento delle immagini (larghezza dell'immagine finale, classi CSS da applicare, eccetera).

`TableOfContents` Questa classe conterrà un elenco degli articoli facenti parte della rivista con le rispettive URL, a partire dal quale verrà realizzato il file JSON necessario al framework grafico.

`ImageResizer` Classe per il ridimensionamento delle immagini: il metodo `Resize(imgPath, widths)` salva sul file system una serie di versioni scalate dell'immagine memorizzata in `imgPath` in base ai contenuti del dizionario `widths`.

`ImageData` Contiene informazioni su un'immagine ottenuta da `ImageResizer` e che saranno necessarie per il successivo inserimento dell'immagine nel proprio articolo.

`ORM` Classe per interagire con la base di dati sottostante.

Ci limiteremo a descrivere nel dettaglio il funzionamento del metodo `NewIssue()` per mezzo di un diagramma di sequenza (figura 3.3).

1. L'oggetto `Service` invia una richiesta al CMS (metodo `FetchCMSResponse()`): la risposta viene deserializzata in un oggetto `Response`.

2. `Service` crea un'istanza della classe `HTMLProcessor`, cui fornisce i dati ottenuti dal CMS, e ne invoca il metodo `CreateHtml()`.

3. Il suddetto metodo di `HTMLProcessor` svolge le seguenti operazioni (contenute nel metodo `CreatePost()`) per ogni oggetto `Post` contenuto in `Response`:

   (a) Legge il contenuto dell'articolo;

   (b) Scarica eventuali immagini e crea diverse versioni di ciascuna secondo quanto contenuto nella proprietà `widths`, usando a tal fine un oggetto `ImageResizer`;

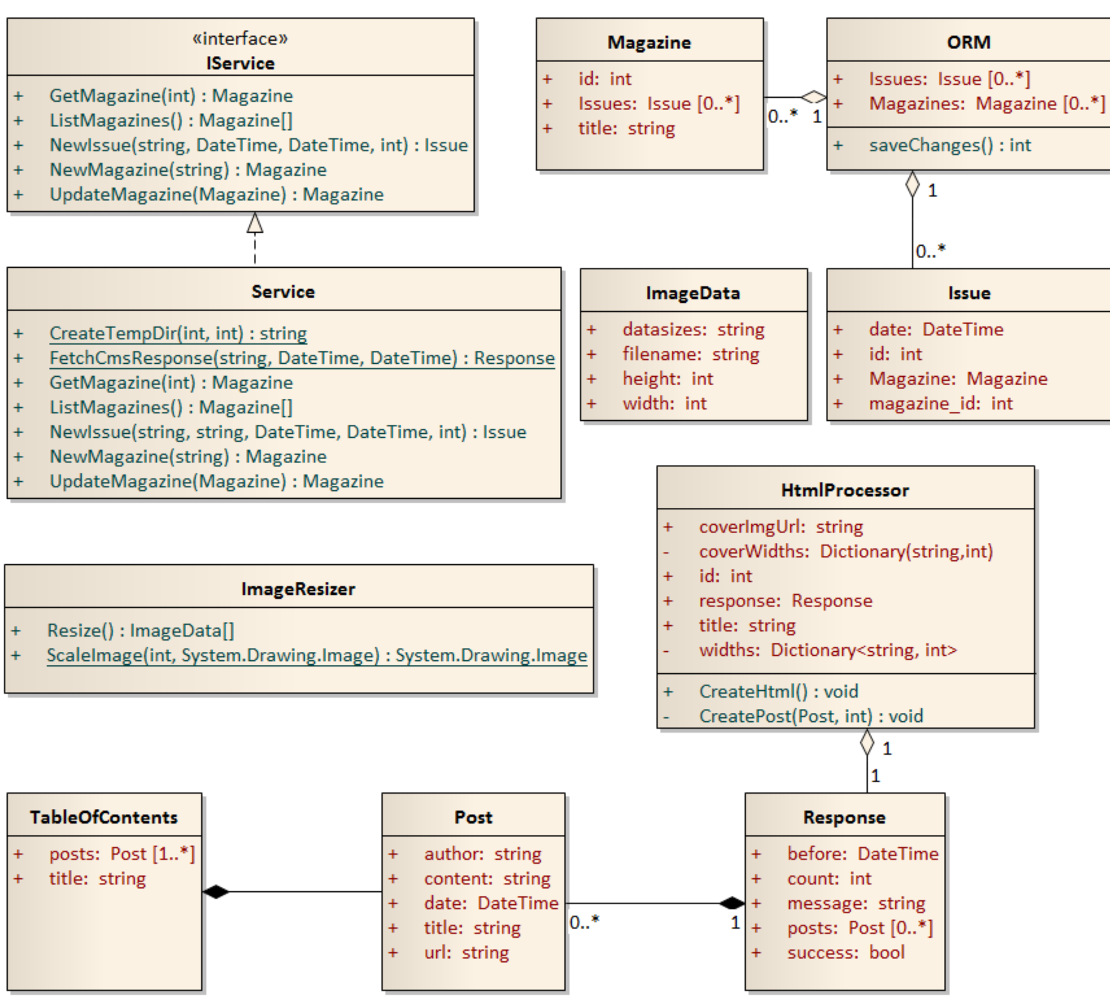


Figura 3.2: Back-end: diagramma delle classi.

   (c) Inserisce le versioni dell'immagine così create nel codice HTML dell'articolo.

   (d) Salva l'articolo su file.

4. `HtmlProcessor` crea l'indice degli articoli contenuti nel nuovo *issue*;

5. `HtmlProcessor` crea il file `index.html`, contenente l'indice ed un'immagine di copertina (che verrà a sua volta ridimensionata in base ai dati contenuti in `coverWidths`).

6. `Service` utilizza la libreria ORM per inserire nella base di dati un nuovo record relativo all'*issue* appena creato.

7. L'oggetto creato viene restituito al front-end in segno di successo.

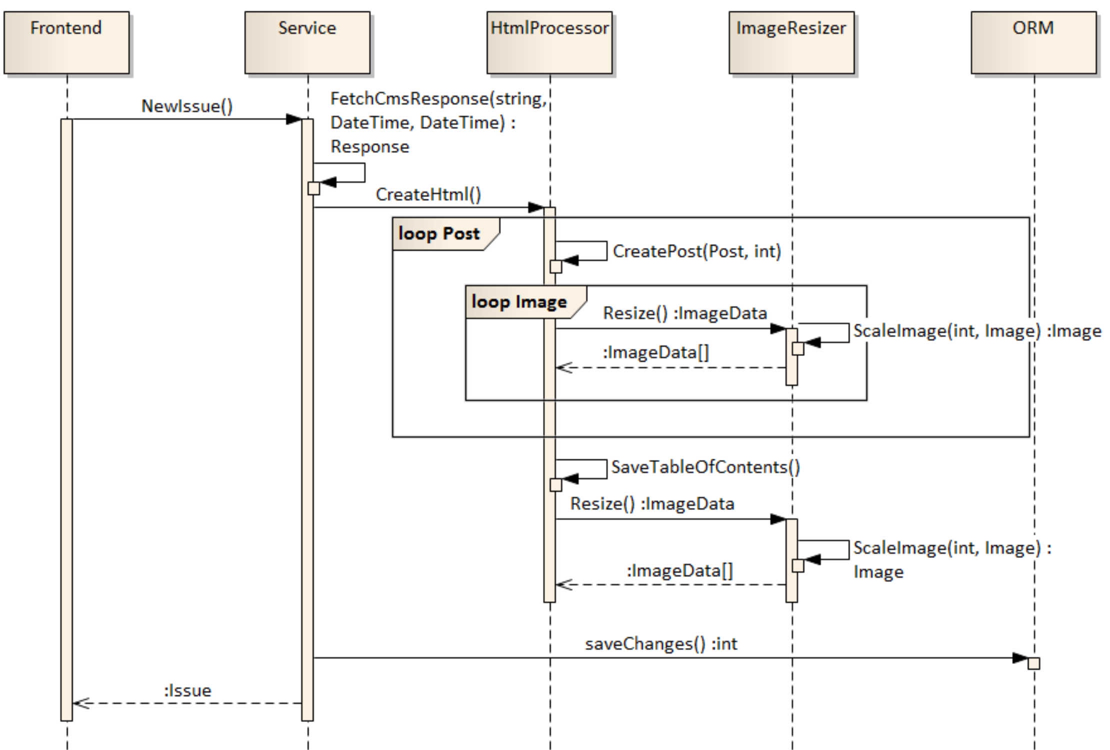


Figura 3.3: Back-end: diagramma di sequenza per il metodo `NewIssue()`.

La directory principale in cui il back-end salva questi dati contiene una sottocartella per ciascun `Magazine`: esse, a loro volta, contengono le directory dei singoli `Issue`. In entrambi i casi useremo l'id dell'oggetto come nome della directory: questo ci permetterà di accedere ad un `Issue` usando solo il suo id e quello del `Magazine` di appartenenza.

3.4 Libreria Data Transfer Object

Per facilitare lo scambio di oggetti tra front-end e back-end (e, come vedremo in seguito, tra web service e applicazione mobile) abbiamo progettato una libreria (figura 3.4) comune alle varie applicazioni e le cui classi verranno usate per serializzare/deserializzare i dati trasferiti via rete.

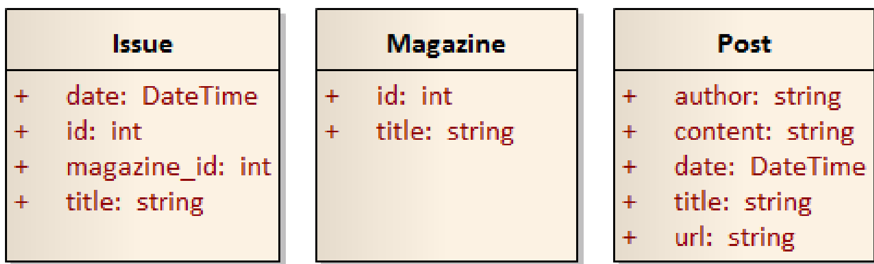


Figura 3.4: Libreria dei Data Transfer Object.

3.5 Web service

Il servizio web (figura 3.5) dovrà fornire agli utenti le funzionalità descritte nel diagramma dei casi d'uso, attraverso una serie di metodi che verranno associati ad altrettanti tipi di richiesta HTTP. Queste funzionalità comprendono:

- Autenticazione: fornendo le proprie credenziali (nome e password) l'utente riceverà un *token* di autenticazione. I token sono un elemento fondamentale dello standard OAuth, che descriveremo nel capitolo seguente.

  Richiesta HTTP: `POST /Token` (il *body* conterrà il nome utente e la password).

- Ritrovamento delle riviste che l'utente ha acquistato.

  Richiesta HTTP: `GET /api/issues`.

- Ritrovamento di un elenco di riviste che l'utente può acquistare.

  Richiesta HTTP: `GET /api/issues/new`.

- Acquisto di una rivista.

  Richiesta HTTP: `POST /api/issues` (il *body* conterrà l'id del `Magazine` e quello dell'`Issue` da acquistare, in formato JSON).

  La risposta del server sarà costituita da un booleano che indichi l'esito dell'operazione e da un eventuale messaggio d'errore (numero già acquistato o inesistente, richiesta incompleta).

Le funzionalità appena elencate richiedono che il servizio recuperi o aggiorni i dati contenuti nel database, il che avverrà attraverso una libreria ORM in modo simile a quanto visto nel back-end.

Sarà inoltre necessario fornire le pagine HTML relative ad una determinata rivista. Questa funzione verrà svolta da un *handler* HTTP: una richiesta all'URL `/static/2/1` restituirà, ad esempio, il file `index.html` della prima rivista appartenente al *magazine* 2.

Per svolgere il proprio compito, l'handler avrà bisogno di informazioni sulla richiesta che gli verranno passate mediante un oggetto `HttpContext`.

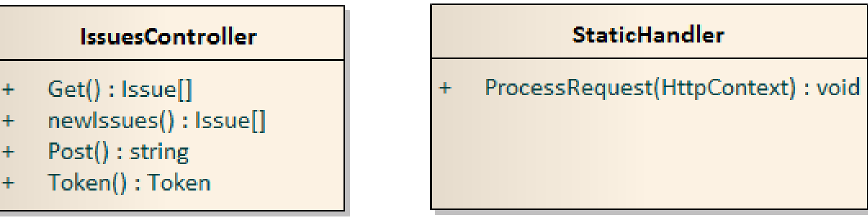


Figura 3.5: Web Service e gestore delle richieste HTTP.

3.6 Applicazioni mobile

Le applicazioni mobile presenteranno le stesse caratteristiche generali:

- Una schermata principale contente la lista degli `Issue` di proprietà dell'utente ed una con quelli acquistabili. Prima di ottenere questi dati, la pagina deve controllare se l'utente è autenticato ed in caso contrario richiedere le credenziali d'accesso

- Una schermata di visualizzazione, che l'utente raggiunge dopo aver selezionato uno tra gli `Issue` disponibili. La pagina deve mostrare i file `.html` relativi all'oggetto selezionato e fornire un menu degli articoli.

I sistemi operativi su cui abbiamo deciso di implementare l'applicazione sono Android 4.0 Ice Cream Sandwich e Windows Phone 8. Piuttosto che sviluppare due applicazioni native completamente separate, abbiamo scelto di adottare la tecnologia Xamarin per riutilizzare buona parte del codice.

Non avremmo potuto ottenere questo risultato senza una chiara distinzione dei ruoli tra le differenti classi che compongono le applicazioni.

Consideriamo una struttura a tre *layer* (figura 3.6), tipica dell'ingegneria del software. Il nostro obiettivo è scrivere una libreria di classi per i due *layer* più lontani dall'utente:

Data layer Gestisce l'accesso ai dati (locali o remoti) e la loro persistenza.

Business layer Descrive le cosiddette "regole di business", cioè i modi in cui l'applicazione può interagire con i dati.

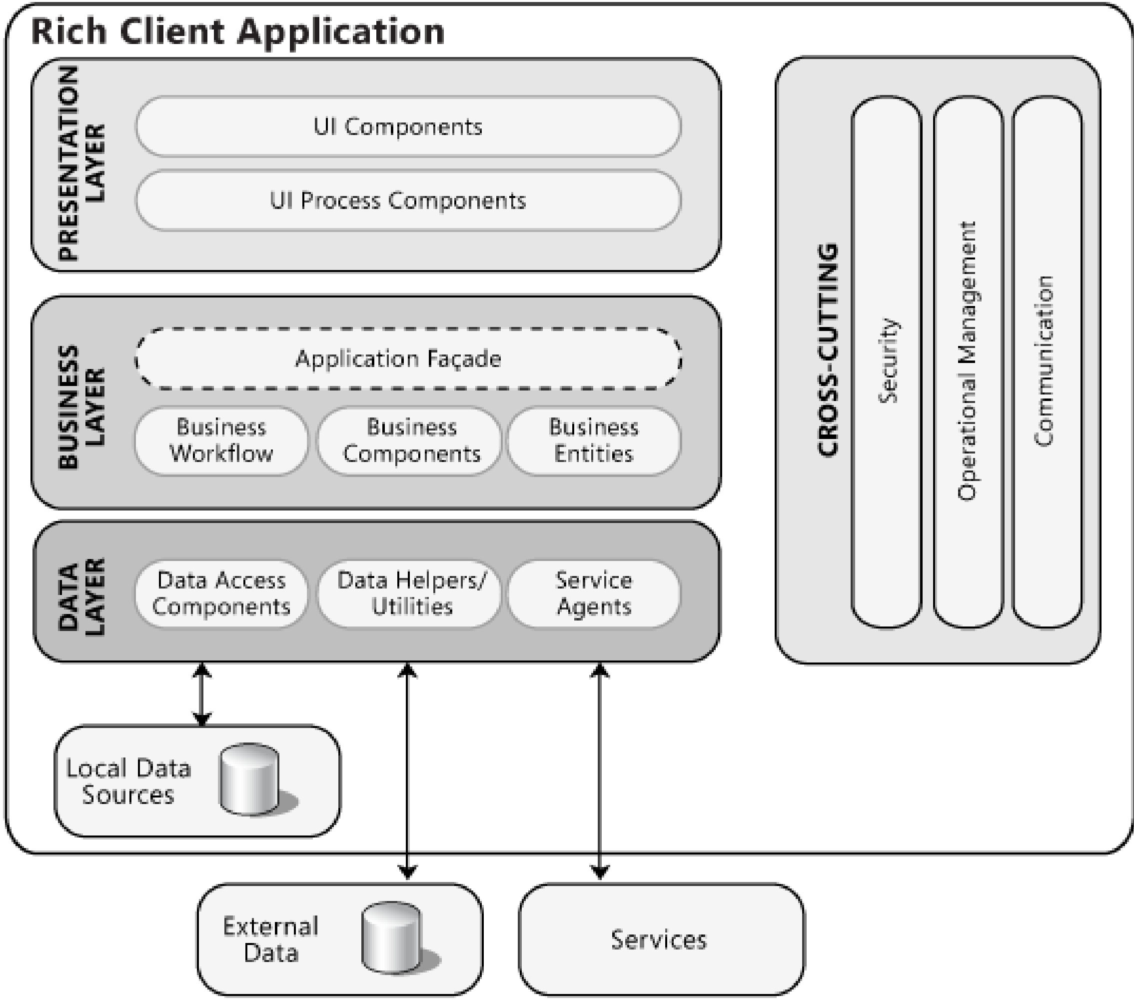


Figura 3.6: Struttura a tre layer di un'applicazione (fonte: Microsoft Patterns & Practices Team [12]).

La libreria in questione è composta di tre classi (figura 3.7) e dovrà essere priva di riferimenti alle API di questo o quel sistema operativo, in modo da poter essere utilizzata sia nell'applicazione Android che in quella Windows Phone.

Le tre classi hanno le seguenti funzioni:

`Token` Deserializza un token di autenticazione.

`Preferences` Fornisce un sistema cross-platform per la persistenza dei dati relativi all'autenticazione (nome dell'utente e token). Useremo il metodo `Save()` per scrivere le proprietà *username* e *token* su un file JSON; al successivo avvio dell'applicazione caricheremo quei dati con il metodo `Load()`.

In questo modo potremo aggirare i differenti sistemi che Google e Microsoft forniscono per risolvere il medesimo problema.

`GutenbergClient` Classe che si occuperà di effettuare le richieste al web service e di deserializzarne le risposte.

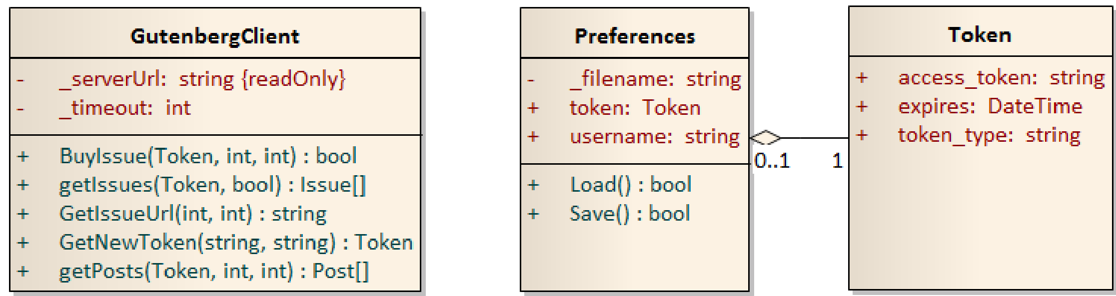
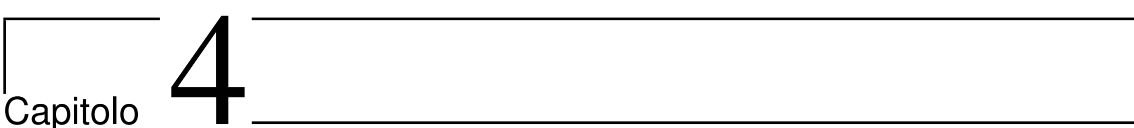


Figura 3.7: Libreria cross-platform per le applicazioni mobile.

I *presentation layer* verranno sviluppati secondo le convenzioni di ciascun sistema operativo.



Tecnologie utilizzate

Riporteremo in questo capitolo alcune informazioni sulle diverse tecnologie utilizzate nell'implementare i componenti della piattaforma.

4.1 CMS: WordPress

WordPress [6] è un Content Management System (CMS) open source, sviluppato in PHP a partire dal 2003. È attualmente uno dei prodotti per il Web più diffusi al mondo [6] per via della semplicità con cui può essere utilizzato ed ampliato per mezzo di plugin.

La principale problematica in cui ci siamo imbattuti durante l'utilizzo di WordPress è la sua architettura poco ordinata, dovuta soprattutto alla volontà di ottenere un prodotto altamente retrocompatibile con i suoi numerosi plugin e temi. Alcuni esempi:

- Massiccio uso di variabili globali;
- Codice prevalentemente procedurale;
- Nessun tipo di astrazione del database: la classe `WP_Query` usata per l'accesso al DB si limita a realizzare stringhe SQL che verranno poi elaborate tramite l'*Original MySQL API* di PHP, recentemente deprecata [7].

4.2 Visualizzazione *responsive*: Treesaver.js

Treesaver [14] è un framework per l'impaginazione di documenti `.html`, pensato per ricreare un'esperienza utente simile a quella delle riviste cartacee. Abbiamo scelto di utilizzare questo framework per la sua compattezza, la sua licenza *open source* e perché riesce ad implementare alcuni capisaldi del *responsive web design* senza ricorrere a funzionalità disponibili solo con gli standard HTML5/CSS3 (che potrebbero non essere del tutto supportati sui dispositivi *mobile*).

Queste caratteristiche, che aiutano a garantire un'esperienza visiva ottimale a prescindere dal dispositivo usato dall'utente, sono le seguenti:

- Utilizzo di una **griglia** HTML le cui singole celle vengono visualizzate, nascoste, riordinate a seconda dello spazio a disposizione.
- **Selective image loading**: il server conterrà più versioni della stessa immagine, differenti tra loro per risoluzione e/o densità di pixel, tra le quali il framework sceglierà la più adatta al dispositivo dell'utente.

Per raggiungere questo risultato, lo script si basa su due file:

- Un "file di risorse" `.html` che descrive il layout di una generica pagina della rivista e delle colonne che la compongono
- Un foglio di stile che definisce le dimensioni minime dei singoli elementi (colonne, immagini, eccetera).

Treesaver potrebbe operare su un singolo file contenente tutti gli articoli della rivista, ma è conveniente separare gli articoli riservando a ciascuno un file `.html`. In tal caso bisognerà fornire a Treesaver un indice in formato JSON, che gli permetterà di caricare i singoli articoli in modo dinamico: quando l'utente raggiungerà l'ultima pagina di un articolo, partirà il caricamento del file successivo.

Il testo dell'articolo andrà inserito in un tag `<article>` nel `body` del proprio file. Treesaver ne valuterà la lunghezza e lo suddividerà in un certo numero di pagine. La struttura della singola pagina verrà scelta tra quelle disponibili nel file delle risorse, in base alle dimensioni dello schermo su cui l'articolo viene visualizzato.

Il tag `<article>` potrà contenere delle immagini, racchiuse in appositi tag `<figure>`. Elencando le diverse versioni dell'immagine lasceremo al framework il compito di selezionare quella più adeguata alla visualizzazione:

```
1 <figure>
2   <img data-sizes="single" data-src="img1.jpg" width="300"
      height="219" />
3   <img data-sizes="double" data-src="img2.jpg" width="600"
      height="439" />
4 </figure>
```

Notiamo che l'attributo `src` è stato sostituito da `data-src`: questo espediente impedisce che il browser carichi automaticamente tutte le immagini menzionate nel codice HTML.

4.3 Framework .NET e Common Language Infrastructure

L'intera piattaforma è basata sul framework .NET, sviluppato da Microsoft a partire dal 2002.

La scelta è stata dettata da due fattori: .NET è la principale tecnologia utilizzata nella sede di sviluppo del progetto ed è particolarmente adatto allo sviluppo di soluzioni di livello aziendale.

Le componenti fondamentali del framework implementano la specifica Common Language Infrastructure (CLI), standardizzata dall'ECMA a partire dal 2001 [5] e dall'ISO/IEC a partire dal 2003 [10].

4.3.1 Compilazione: Common Intermediate Language

I compilatori dei linguaggi CLI-compatibili generano istruzioni in un linguaggio intermedio, il Common Intermediate Language; questo codice viene infine convertito in linguaggio macchina da una *virtual machine*. Le modalità di funzionamento della *virtual machine* possono variare a seconda dell'implementazione: la *Common Language Runtime* di Microsoft effettua una compilazione *just-in-time* del codice, mentre la Mono Runtime offre anche la possibilità di compilare *ahead-of-time* ed ottenere un eseguibile dipendente dalla piattaforma.

4.4 Il linguaggio C#

C# è un linguaggio orientato agli oggetti, sviluppato da Microsoft e descritto negli standard ECMA-334 [4] e ISO/IEC 23270:2006 [9]. Lo sviluppo del linguaggio si prefigge determinati obiettivi, chiaramente espressi nelle specifiche appena citate:

- Codice altamente portabile ed eseguibile su sistemi sia *hosted* che *embedded*;
- Elevata "Programmer portability", cioè facilità di utilizzo da parte del programmatore a prescindere dalla piattaforma di sviluppo;
- Applicazioni parsimoniose in termini di memoria e potenza di calcolo[1];
- Supporto allo sviluppo di software da utilizzare in ambienti distribuiti.

4.4.1 Caratteristiche del linguaggio

Sebbene una descrizione esaustiva del linguaggio sarebbe fuori luogo, è necessario descrivere alcune peculiarità che ricorreranno negli stralci di codice presenti in questa tesi.

Typing implicito C# è staticamente tipato: il tipo delle variabili va dichiarato e viene controllato in fase di compilazione[2]. La parola chiave `var` corrisponde ad una dichiarazione *implicita* del tipo, che viene cioè determinato dal compilatore in base al valore iniziale della variabile. I vantaggi del typing implicito (del quale, ad ogni modo, è meglio non abusare) possono apparire più chiari mediante questo esempio:

```
1 Dictionary<string, string> obj = new
     Dictionary<string,string>();
2 var obj_1 = new Dictionary<string, string>();
```

Gli oggetti `obj` e `obj_1` sono entrambi istanze della classe `Dictionary <string, string>`, ma la seconda definizione è più compatta e leggibile.

[1] Non si intende, in questo ambito, entrare in diretta competizione con il C o i linguaggi assemblativi.
[2] Se necessario, si può usare il tipo `dynamic` per fare in modo che questi controlli vengano effettuati solo a run-time.

Tipi anonimi Un insieme di dati può essere incapsulato in un oggetto di "tipo anonimo", derivato direttamente dalla classe `Object` (che si trova al vertice della gerarchia dei tipi del framework .NET).

```
1 var anonymous = new { firstName = "John", lastName = "Doe" };
2 System.Console.WriteLine(anonymous.firstName) // John
```

L'oggetto che viene costruito sarà in sola lettura: nel nostro esempio, un'istruzione come `anonymous.firstName="Jane"` causerebbe un errore di compilazione.

Attributi In C#, gli attributi sono **metadati** applicati per descrivere alcune caratteristiche di un "blocco di codice" (ad esempio una classe o un metodo) e che saranno accessibili sia in fase di compilazione che a run-time.

Consideriamo ad esempio `ObsoleteAttribute`, uno degli attributi predefiniti in .NET: lo si può utilizzare per segnalare allo sviluppatore che un elemento (classe, metodo, struttura dati. . .) non deve più essere utilizzato.

```
1 public class MyClass {
2         [System.Obsolete("use NewMethod", error: true)]
3         public void OldMethod1() { ... }
4         [System.Obsolete("use NewMethod", error: false)]
5         public void OldMethod2() { ... }
6         public void NewMethod() { ... }
7 }
```

Una chiamata al metodo `C.OldMethod1()` provocherà un errore di compilazione, mentre `C.OldMethod2()` genererà soltanto un avviso. In entrambi i casi, il compilatore restituirà allo sviluppatore il messaggio definito all'interno dell'attributo (nell'esempio, "`use NewMethod`").

Proprietà e campi Un campo (*field*) è una variabile dichiarata all'interno di una classe. Per garantire l'incapsulamento, i campi non dovrebbero mai essere direttamente accessibili dall'esterno della propria classe: l'esposizione avviene per mezzo delle proprietà (*properties*).

Una proprietà è formata da due funzioni d'accesso (*accessors*), `get` e `set`, che definiscono il modo in cui il codice esterno alla classe può leggere o scrivere il campo da essa esposto: essi, in altre parole, svolgono le operazioni che altri linguaggi affidano ai metodi *getter* e

setter. Una proprietà è *read-only* o *write-only* se la funzione `set` o `get` viene omessa.

All'esterno, la proprietà appare indistinguibile da un *public member*.

C# 3.0 ha introdotto la possibilità di dichiarare proprietà implementate automaticamente, cioè prive di campi esplicitamente definito: il compilatore provvederà a creare campi anonimi ad esse associati.

```
public class RationalNumber {
  // Il numeratore puo' essere un intero qualsiasi: possiamo
     usare una proprieta' auto-implemented
  public int Numerator{ get; set; }
  // Il denominatore deve essere diverso da zero
  private int _den;
  public int Denominator {
    // int x = rational.Denominator;
    get {
      return _den;
    }
    // rational.Denominator = 1
    set {
      if (value != 0) {
        _den = value;
      }
      else {
        throw new System.ArgumentOutOfRangeException();
      }
    }
  }
  // Un esempio di proprieta' read-only il cui valore non e'
     legato ad un campo
  public double Value {
    get {
      return (double) Numerator / Denominator;
    }
  }
}
```

`async` e `await` C# supporta nativamente la programmazione asincrona. Un metodo dotato del modificatore `async` può essere eseguito in maniera asincrona utilizzando l'operatore `await` (naturalmente, anche il metodo che contiene l'istruzione `await` dovrà essere dichiarato asincrono).

Un metodo `async` restitisce valori di tipo `Task<T>` dove `T` è il tipo di dato da restituire (`Task` se il metodo non restituisce alcun valore), oppure `void`.

Questa funzionalità è risultata particolarmente utile nello sviluppo delle applicazioni *mobile*: un'implementazione sincrona dei metodi di comunicazione con il server avrebbe bloccato il thread dell'interfaccia grafica, peggiorando la *user experience* dell'applicazione.

Eventi e delegati Un delegato (*delegate*) è un tipo che permette allo sviluppatore di chiamare un metodo in modo indiretto e di scrivere metodi anonimi. In pratica offrono la flessibilità dei puntatori a funzione senza i rischi di *type safety* ad essi collegati: ciò li rende molto utili per definire funzioni di *callback*.

```
1  delegate void Callback(int x);
2  class Calculator {
3    public static void Sum(int x, int y, Callback c){
4      c(x + y);
5    }
6  }
7
8  // Nel metodo Main()
9  Callback c = delegate(int x) { Console.WriteLine(x); };
10 Calculator.Sum(11, 4, c); //Stampa "15" sulla console
```

Se un oggetto (ad esempio, un componente di un'interfaccia grafica) espone un *gestore* per un determinato evento, possiamo usare uno o più delegati per specificare quali operazioni effettuare quando quell'evento si verifica:

```
1    button.Click += delegate {
2      Console.WriteLine("Hai fatto click sull'oggetto button");
3    }
```

Risorse non gestite e l'istruzione `using` Le *virtual machine* su cui il codice C# (compilato in CLI) verrà eseguito si basano su una memoria *heap* il cui contenuto viene gestito da un *garbage collector*.

Le risorse che non possono essere allocate nell'*heap* sono dette "non gestite" (*unmanaged resources*), cioè fuori dalla portata diretta del *garbage collector*.

Un oggetto che deve far riferimento a risorse non gestite viene convenzionalmente realizzato come implementazione dell'interfaccia `System.IDisposable` e le modalità di deallocazione di tali risorse vengono descritte nel metodo `Dispose()`.

Il modo più corretto di utilizzare questi oggetti è mediante l'istruzione `using`: il compilatore farà sì che `Dispose()` venga chiamato al termine del blocco, anche in caso di eccezione.

I due blocchi di codice nel prossimo esempio sono concettualmente equivalenti, ma il secondo è più chiaro e limita lo *scope* dell'oggetto al blocco `using`.

```
1  /* Blocco 1 */
2  var disp = new DisposableObject();
3  try {
4    disp.Method();
5  }
6  catch {
7   handleException();
8  }
9  finally {
10   disp.Dispose();
11 }
12 // disp e' definita nello scope "principale".
13 /* Blocco 2 */
14 using (var disp = new DisposableObject()){
15   try {
16     disp.Method();
17   }
18   catch {
19    handleException();
20   }
21 }
22 // disp non e' definita al di fuori del blocco.
```

4.5 Mono e Xamarin

4.5.1 Il progetto Mono

Il progetto Mono [15] è un'implementazione *open source* della specifica CLI che consente l'esecuzione di codice .NET anche sui sistemi UNIX e

BSD. Molti dei principali sviluppatori di Mono sono confluiti nella società Xamarin, che attualmente ne guida lo sviluppo.

Xamarin fornisce i *port* di Mono per i sistemi Android e iOS, e altre librerie che replicano le API dei due sistemi in linguaggio C#[3]. L'applicazione non è stata implementata su sistemi iOS per l'assenza di una postazione Apple nella sede di Carsoli, ma vedremo che le scelte architetturali che abbiamo seguito renderanno più semplice lo sviluppo di un futuro *port* per questo sistema operativo.

4.5.2 Architettura di un'applicazione Xamarin

Xamarin non si propone, al momento, come un vero e proprio framework di sviluppo cross-platform, ma come una soluzione per aumentare la condivisione di codice tra applicazioni destinate a dispositivi diversi.

Lo sviluppatore potrà infatti implementare la logica e l'accesso ai dati della propria applicazione in modo che il codice sia compatibile con tutti i sistemi operativi supportati da Xamarin, per poi realizzare i diversi *presentation layer* in base agli standard delle singole piattaforme. I *presentation layer* dovranno essere realizzati separatamente: lo sviluppo per Windows Phone è analogo a quello di un'applicazione nativa, mentre quello per Android avviene per mezzo di apposite librerie che consentono lo sviluppo in C# anziché Java.

Questa scelta architetturale comporta un carico di lavoro leggermente superiore rispetto ad alternative completamente cross-platform, ma consente anche di seguire più accuratamente le HIG[4] tracciate dai produttori dei sistemi operativi, garantendo un *look and feel* più vicino a quello delle applicazioni native.

4.6 Altre tecnologie

4.6.1 ADO.NET Entity Framework

Entity Framework (EF) è una tecnologia di *object-relational mapping*: il suo scopo è semplificare l'integrazione tra codice orientato ad oggetti e basi di

[3] È importante notare che i pacchetti offerti da Xamarin non sono open-source, a differenza del progetto Mono su cui si basano.

[4] Human Interface Guidelines.

dati relazionali. Si basa su ADO.NET, un insieme di tecnologie per l'accesso a diversi tipi di sorgenti dati.

Nell'approccio *Database First* il programmatore definisce una serie di tabelle sul database, lasciando ad Entity Framework il compito di generare le classi con cui potrà interagire all'interno del codice dell'applicazione.

L'approccio *Code First* è speculare: Entity Framework analizza le classi create dal programmatore e realizza le tabelle necessarie alla persistenza degli oggetti.

4.6.2 Windows Communication Foundation

Windows Communication Foundation (WCF) è un framework per realizzare applicazioni distribuite e orientate ai servizi.

Un *servizio* è un componente software che effettua operazioni in base a richieste giunte dall'esterno. Nella maggior parte dei casi l'invio di richieste avviene attraverso la rete Internet: si parla quindi di servizi web (*web services*).

Un'architettura orientata ai servizi (SOA) può portare diversi vantaggi rispetto allo sviluppo di una soluzione software "monolitica": i singoli servizi sono facilmente scalabili e riutilizzabili, indipendentemente dai linguaggi di programmazione usati nelle applicazioni che li interrogano.

Un servizio WCF è raggiungibile tramite un *endpoint* caratterizzato da tre elementi:

Address Descrive l'indirizzo ed il protocollo da utilizzare per contattare il servizio. Alcuni esempi:

`http://localhost:8001` (HTTP)
`net.pipe://localhost/piped` (*named pipe*)

Binding Fornisce informazioni aggiuntive sulle modalità di comunicazione con il servizio: codifica, autenticazione, sicurezza, ecc.

Contract Descrive l'insieme di operazioni offerte dal servizio.

Il servizio potrà essere eseguito in varie modalità, ad esempio come servizio Windows o all'interno di IIS (il web server sviluppato da Microsoft).

Noto l'indirizzo dell'endpoint, Visual Studio può generare un *riferimento al servizio* all'interno di un'applicazione separata: si potranno effettuare le operazioni esposte nel Contract sotto forma di chiamate ai metodi di una classe.

4.6.3 OAuth

I metodi del web service dovranno essere accessibili solo ad utenti autorizzati. Il processo di autorizzazione segue lo standard aperto OAuth 2.0, che si basa sull'utilizzo dei cosiddetti *bearer token* [11].

Un *bearer token* è una stringa generata da un server di autorizzazione e valida per un periodo di tempo limitato: chiunque sia in possesso del token sarà autorizzato ad accedere alle risorse finché esso non scade (da cui il nome: letteralmente, "token valido al portatore"). Per ottenere un token valido, l'utente deve:

1. Registrarsi presso il proprietario della risorsa, scegliendo un nome utente e una password;

2. Inviare le proprie credenziali al server di autorizzazione, che risponderà con un token valido (specificandone anche il tempo di validità).

Per accedere alla risorsa, l'utente dovrà inserire il token in un campo `Authorization` nell'header della richiesta, specificando anche il tipo di autorizzazione (`Bearer`). Ad esempio:

```
1 GET /resource HTTP/1.1
2 Host: server.example.com
3 Authorization: Bearer mF_9.B5f-4.1JqM
```

Il server della risorsa comunicherà con il server di autorizzazione per verificare la validità del token e solo in tal caso restituirà la risorsa all'utente.

Capitolo 5

Implementazione

5.1 CMS: plugin di integrazione per WordPress

WordPress segue, in linea di massima, un design *event-driven*: ogni volta che il server riceve una richiesta, essa scatena una sequenza di eventi (detti *azioni*). Un plugin di WordPress consiste in un insieme di funzioni PHP che lo sviluppatore associa ad una o più azioni.

```
1 function hello_world() {
2   echo "Hello World";
3 }
4 add_action('action', 'hello_world');
```

Gli script fondamentali di WordPress conterranno istruzioni `do_action('action')` che eseguiranno le funzioni associate alle azioni.

Uno script in particolare, situato in `wp-admin/admin-ajax.php`, fornisce un endpoint particolarmente adatto a richieste AJAX. In questo caso, infatti, l'esecuzione del plugin avverrà a seguito di un caricamento "minimale" delle funzionalità di WordPress, tralasciando cioè tutti i componenti relativi all'interfaccia utente del sito.

```
1 function gutenberg_response() {
2   if (!gutenberg_is_request_valid()) { // Validazione delle date
3     $response = array(
4          'success' => false,
5          'message' => "Richiesta non corretta."
6     );
7   }
8   else {
```

```
9       // Parametri della query
10      $args = array(
11          'date_query' => array(
12              array(
13                  'before'    => $_REQUEST['before'],
14                  'after'     => $_REQUEST['after'],
15                  'inclusive' => true,
16              ),
17          ),
18      );
19      $query = new WP_Query($args);
20      $response = array(
21        'success'   => true,
22        'message'   => null,
23        'before'    => $_REQUEST['before'],
24        'after'     => $_REQUEST['after'],
25        'count'     => $query->found_posts,
26        'posts'     => array()
27      );
28      if ($query->found_posts == 0) {
29        $response['message']= 'Nessun post per il periodo
            richiesto.';
30      }
31      while ($query->have_posts()) {
32        $query->the_post();
33        $response[ 'posts' ][] = array(
34          'title' => $query->post->post_title,
35          'author' => get_the_author(),
36          'content' => wpautop($query->post->post_content),
37          'date' => get_the_date('Y-m-d')
38        );
39      }
40    }
41    header('Content-Type: application/json; charset=utf8');
42    echo json_encode( $response );
43    die();
44  }
45  add_action('wp_ajax_gutenberg', 'gutenberg_response');
46  add_action('wp_ajax_nopriv_gutenberg', 'gutenberg_response');
```

I due `add_action()` posti in fondo al file del plugin associano la funzione alle richieste di tipo `/wp-admin/admin-ajax.php?action=gutenberg`: la prima azione viene eseguita se l'utente che invia la richiesta ha effettuato

il login, mentre la seconda fornisce le stesse funzionalità agli utenti non autenticati.

5.2 Base di dati

Siamo partiti dalla progettazione delle tabelle contenute in un database SQL Server e destinate a rappresentare gli oggetti `Magazine` e `Issue` (figura 5.1).

Notiamo che la chiave primaria della tabella `Issues` è composta da due campi: `id` e `magazine_id`, che è anche una *foreign key* che fa riferimento all'id di un record della tabella `Magazines`.

Figura 5.1: Diagramma delle tabelle in SQL Server.

5.3 Sistema di produzione

5.3.1 Libreria Data Transfer Object

La libreria `GutenbergDTOLibrary` contiene oggetti che verranno utilizzati per trasferire dati tra front-end e back-end.

Utilizzeremo gli attributi `[DataContract]` e `[DataMember]` per fare in modo che WCF serializzi e deserializzi i dati durante il loro trasferimento. Questa procedura è necessaria perché, come abbiamo detto in precedenza, l'endpoint di un servizio WCF (come il nostro back-end) deve fornire un *Contract* contenente le operazioni offerte dal servizio e i tipi di dati restituiti o richiesti come parametri.

```
1 [DataContract]
2 public class Issue
3 {
4   [DataMember]
5   public int id { get; set; }
```

```
6   [DataMember]
7   public int magazine_id { get; set; }
8   [DataMember]
9   public DateTime date { get; set; }
10  [DataMember]
11  public string title { get; set; }
12 }
13 [DataContract]
14 public class Magazine
15 {
16   [DataMember]
17   public int id { get; set; }
18   [DataMember]
19   public string title { get; set; }
20 }
```

5.3.2 Back-end

Il back-end è stato realizzato sotto forma di servizio WCF ospitato su server IIS. Abbiamo descritto le operazioni fornite dal servizio in un'interfaccia `IService` che è stata poi implementata da una classe `Service`.

L'interfaccia sarà il Contract del servizio: i metodi dotati dell'attributo `OperationContract` saranno accessibili tramite WCF, mentre l'attributo `FaultContract` indicherà la possibilità che un metodo lanci un'eccezione che verrà inoltrata al client del servizio.

```
1  [ServiceContract]
2  public interface IService
3  {
4    [OperationContract]
5    [FaultContract(typeof(string))]
6    List<GutenbergDTOLibrary.Magazine> ListMagazines();
7
8    [OperationContract]
9    [FaultContract(typeof(string))]
10   GutenbergDTOLibrary.Magazine GetMagazine(int id);
11
12   [OperationContract]
13   [FaultContract(typeof(string))]
14   GutenbergDTOLibrary.Magazine
        UpdateMagazine(GutenbergDTOLibrary.Magazine m);
15
```

```
16
17    [OperationContract]
18    [FaultContract(typeof(string))]
19    GutenbergDTOLibrary.Magazine NewMagazine(string title);
20
21    [OperationContract]
22    [FaultContract(typeof(string))]
23    GutenbergDTOLibrary.Issue NewIssue(int magazineId, DateTime
         before, DateTime after, string url, string coverImgUrl);
24  }
```

Dopo aver fornito a Visual Studio le credenziali di accesso a SQL Server, abbiamo selezionato le tabelle `Issues` e `Magazines` lasciando ad Entity Framework il compito di creare le classi con cui manipoleremo il contenuto del database. Abbiamo così ottenuto due classi `Issue`, `Magazine` che rappresenteranno un singolo record ed una terza, che abbiamo chiamato `GutenbergEntities`, con la quale potremo effettuare query o salvare nuovi record.

Il metodo `UpdateMagazine`, che modifica il titolo di una testata, fornisce un buon esempio delle funzionalità di questa classe:

```
1  public GutenbergDTOLibrary.Magazine
      UpdateMagazine(GutenbergDTOLibrary.Magazine m)
2  {
3    using (var db = new gutenbergEntities())
4    {
5      Magazine magazine = db.Magazines.Find(m.id);
6      if (string.IsNullOrEmpty(m.title.Trim()))
7      {
8        throw new FaultException("Manca il titolo.");
9      }
10     magazine.title = m.title = m.title.Trim();
11     db.Entry(magazine).State =
          System.Data.Entity.EntityState.Modified;
12     db.SaveChanges();
13     return m;
14   }
15 }
```

Il metodo riceve un oggetto `m: GutenbergDTOLibrary.Magazine` dal front-end ed usa `db: GutenbergEntities` per ritrovare il corrispondente record nel database (riga 5).

Questo record viene “mappato” su un oggetto `magazine: Magazine`, il cui titolo viene sostituito con quello fornito dal front-end (riga 11).

Infine, segnaliamo che lo stato dell’entità `magazine` è stato modificato e salviamo i cambiamenti sul database (righe 11 e 12).

Le credenziali e l’indirizzo del server SQL non verranno incorporate nell’applicazione: saranno invece memorizzate in un file di configurazione che potrà essere modificato anche in seguito alla compilazione del codice.

5.3.3 Front-end

Nella realizzazione del front-end abbiamo utilizzato la tecnologia ASP.NET MVC, pensato per velocizzare la realizzazione di applicazioni web secondo il pattern Model-View-Controller.

Abbiamo poi inserito le informazioni di accesso al back-end (che nel frattempo è stato pubblicato su un server IIS di prova) nel file XML di configurazione `Web.config`.

```
1  <client>
2    <endpoint address="http://localhost/Backend/Service.svc"
        binding="basicHttpBinding"
3      bindingConfiguration="BasicHttpBinding_IService"
          contract="BackendReference.IService"
4      name="BasicHttpBinding_IService" />
5  </client>
```

Vediamo, ad esempio, il funzionamento del metodo che si occupa di visualizzare un form di inserimento di un nuovo *issue*. I metodi di ciascun controller vengono automaticamente associati ad URL nel formato `/{controller}/{metodo}`, quindi una richiesta a `/Issue/Insert` causerà l’esecuzione di `IssueController.Insert()`:

```
1  public ActionResult Insert() {
2    BackendReference.ServiceClient service = new
        BackendReference.ServiceClient();
3    try {
4      var magazines = service.ListMagazines();
5      service.Close();
6      ViewData["Magazines"] = new SelectList(magazines, "id",
          "title");
7    }
8    catch (Exception e) {
9      service.Close();
```

```
10          ViewData["Magazines"] = new SelectList("");
11          ViewData["error"] = e.Message;
12      }
13      Models.IssueForm i = new Models.IssueForm();
14      return View(i);
15  }
```

L'elenco dei *magazine* disponibili viene caricato con l'istruzione `service.ListMagazines()`, lasciando a WCF il compito di effettuare la richiesta via rete e deserializzare i dati ricevuti.

`ViewData` è un array associativo i cui contenuti potranno essere visualizzati dalla *view* del metodo.

`IssueForm` rappresenta il form: specifica di quali campi è composto, quali di essi sono obbligatori e gli eventuali metodi di validazione da applicare a ciascuno di essi.

Quando l'editore clicca sul tasto di invio del form, i dati vengono inviati con metodo `POST` alla stessa URL: useremo quindi l'attributo `HttpPost` per distinguere il metodo incaricato di processare tali dati.

```
1  [HttpPost]
2  public ActionResult Insert(int magazine_id, string url, string
       coverImgUrl, string before, string after) {
3    // before e after vangono convertiti in
4    // due valori DateTime, 'datetime_before' e 'datetime_after'
5    BackendReference.ServiceClient service = new
         BackendReference.ServiceClient();
6    try {
7        var issue = service.NewIssue(magazine_id, datetime_before,
             datetime_after, url, coverImgUrl);
8        service.Close();
9        TempData["message"] = String.Format("Rivista creata
             correttamente: {0}, n. {1}", issue.title, issue.id);
10   }
11   catch (Exception e) {
12       service.Close();
13       TempData["error"] = e.Message;
14   }
15   return RedirectToAction("Index");
16 }
```

Stavolta abbiamo usato `TempData` anziché `ViewData` per mantenere i dati in memoria anche dopo il *redirect* finale.

5.4 Web service

5.4.1 Microsoft Web API

Il web service vero e proprio si basa sulla tecnologia Web API di Microsoft, che consente di realizzare servizi web accessibili via HTTP.

Il funzionamento di questo framework è molto simile a quanto visto in ambito ASP.NET MVC, con l'ovvia differenza che non avremo classi *view*: i dati restituiti da un metodo saranno direttamente restituiti all'utente in formato JSON.

Inoltre, Web API implementa un sistema di autenticazione basato sullo standard OAuth: per registrare un utente, ad esempio, è sufficiente inviare una richiesta `POST` all'indirizzo `/api/Account/Register`, includendo nel *body* della richiesta le sue credenziali (nome utente e password).

Analogamente, l'utente può ricevere un token inviando il proprio nome utente e la relativa password all'indirizzo `/Token`. In ambito produttivo sarà necessario limitare queste potenzialità (ad esempio, consentendo l'accesso all'operazione `Register` solo ad utenti amministratori) e configurare il server per comunicare via HTTPS.

Per fornire queste operazioni l'applicazione poggia su diverse tabelle nel nostro database, create da Visual Studio dopo l'inizializzazione del progetto: la più importante tra queste è `AspNetUsers`, che contiene l'id, il nome e la password (opportunamente criptata) di ciascun utente.

Il sistema è pensato per non memorizzare i token in nessun luogo: il server li crea crittografando alcune informazioni sull'utente e la data di creazione del token stesso, e di conseguenza considera validi solo quelli che possono essere decriptati con la propria chiave privata.

Nell'implementare le funzionalità previste in fase di progettazione, abbiamo per prima cosa creato una tabella `Users_Issues` per implementare la relazione (di tipo molti-a-molti) tra utenti e riviste: ciascun record collegherà un utente ad un *issue* in suo possesso (figura 5.2).

Anche in questo caso, abbiamo fatto ricorso a Entity Framework per realizzare classi che ci permettessero di interagire con la base dati mediante codice orientato agli oggetti.

Abbiamo poi fornito alla classe `IssuesController` l'attributo `[Authorize]`. Ciò significa che, in fase di run-time, il server valuterà le richieste indirizzate ai metodi di questo controller e bloccherà quelle prive di un token valido.

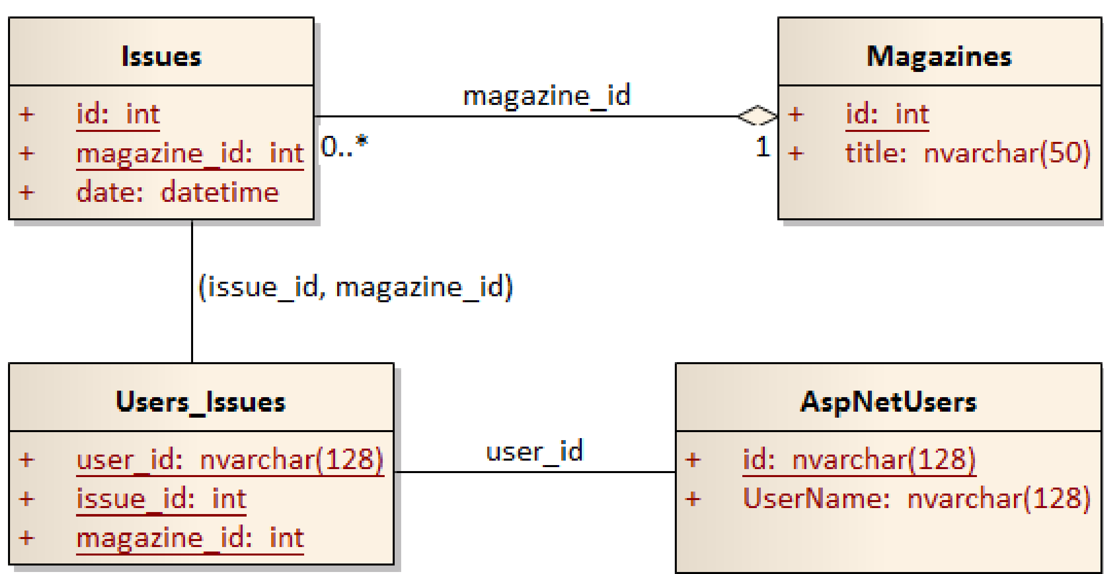


Figura 5.2: Diagramma delle tabelle utilizzate dal web service.

I metodi `Get()` e `NewIssues()` si limitano ad effettuare una query su `Users_Issues` e restituire gli *issue* in possesso (o meno) dell'utente.

Leggermente più elaborato è il metodo `Post()`, che gestisce la procedura di "acquisto" di un nuovo *issue*. La richiesta deve contenere un oggetto JSON contenente i dati per identificare l'*issue*, cioè `id` e `magazine_id`. Il server deve effettuare diversi controlli sull'oggetto ed inviare un messaggio d'errore se l'utente ha specificato un numero inesistente o già in suo possesso, oppure se i dati inviati sono errati o incompleti. Vediamo uno di questi controlli:

```
// L'oggetto User viene creato dal server in base al token
// fornito nella richiesta
string uid = User.Identity.GetUserId();
int count = db.Users_Issues.Where(
  x => x.user_id == uid &&
  x.issue_id == json.id &&
  x.magazine_id == json.magazine_id
  ).Count();
if (count > 0) {
  return new {
    success = false,
    message = "Questo elemento e' gia' in tuo possesso."
  };
```

```
14 }
```

Se l'utente ha inviato una richiesta valida, il metodo dovrà invece aggiungere un record alla tabella `Users_Issues` e rispondere con un messaggio di conferma:

```
try {
    db.Users_Issues.Add(new Users_Issues()
        {
            user_id = uid,
            issue_id = json.id,
            magazine_id = json.magazine_id
        });
    db.SaveChanges();
    return new { success = true, message = "Elemento
       acquistato." };
}
catch {
  return new {
    success = false,
    message = "Errore durante l'acquisto."
  };
}
```

5.4.2 Handler HTTP

Il componente gestirà anche l'accesso ai file che compongono i diversi *issue* (file `.html`, immagini, eccetera).

Queste operazioni sono analoghe a quelle effettuate da un comune web server: abbiamo quindi evitato di implementarle attraverso la piattaforma Web API, limitandoci a personalizzare il funzionamento del server IIS in base alle nostre necessità.

Abbiamo cioè realizzato un gestore (*handler*) HTTP personalizzato e, modificando la configurazione del componente, gli abbiamo affidato l'elaborazione di tutte le richieste HTTP il cui percorso contenga la directory `static`.

```
<add name="static" path="static" verb="*"
   type="WebService.StaticHandler, WebService"
   resourceType="Unspecified" />
```

Il file system non contiene una vera e propria directory `static`: l'*handler* si occuperà di recuperare le risorse dalla directory in cui il back-end le aveva memorizzate.

L'*handler* opera nel modo seguente:

1. Se l'autenticazione è obbligatoria, recupera le informazioni sull'utente che ha effettuato la richiesta e risponde con un codice `401 Unauthorized` se l'utente non ha fornito un token valido o se non è in possesso dell'*issue* richiesto.

2. Calcola il percorso fisico della risorsa richiesta e:

   - Se l'utente aveva richiesto una directory esistente, invia una risposta `301 Moved Permanently` e reindirizza l'utente verso il file `index.html` di quella directory. Ad esempio, una richiesta verso `/static/1/1` genera un *redirect* verso `/static/1/1/index.html`.
   - Se l'utente aveva richiesto un file, si comporta come un normale server web: la risposta avrà codice `200` e conterrà il file richiesto se esso è presente nel file system, altrimenti il codice di risposta sarà `404 Not Found`.

La directory del file system contenente gli *issue* ed il booleano che impone l'autenticazione degli utenti sono memorizzate su un file di configurazione, accessibile anche dopo la compilazione del codice.

Abbiamo inserito la possibilità di disabilitare il processo di autenticazione per semplificare lo sviluppo e il *debug* dell'*handler* stesso e delle applicazioni *mobile* che con esso dovranno interagire.

5.5 Applicazioni Mobile

Abbiamo preferito iniziare lo sviluppo *mobile* a partire dal sistema operativo Android, per verificare l'integrazione tra le librerie Xamarin e le API fornite da Google. Una volta giunti ad uno stadio abbastanza maturo dello sviluppo, abbiamo dato inizio alla fase di implementazione su Windows Phone: come avevamo previsto, il modello di sviluppo che abbiamo scelto ci ha consentito di riutilizzare gran parte del codice su entrambe le piattaforme.

5.5.1 Libreria cross-platform GutenbergCore

Il *namespace* GutenbergCore contiene l'insieme delle classi utilizzate in entrambe le applicazioni: come si è detto, questa libreria descriverà il *data layer* ed il *business layer* del software.

Data Layer e persistenza dei dati

Il *data layer* è costituito da una serie di classi (Issue, Post, Token) che modellano i dati scambiati tra l'applicazione ed il web service; le modalità con cui avvengono questi scambi fanno invece parte del business layer.

Altra componente importante è quella relativa alla persistenza dei dati stessi: i due sistemi operativi presi in esame forniscono, infatti, soluzioni differenti per accedere alla memoria di massa del dispositivo. Abbiamo quindi realizzato una classe Preferences capace di svolgere queste operazioni in maniera indipendente dal sistema operativo sottostante.

Abbiamo accennato in precedenza che la classe salverà e caricherà i dati attraverso l'uso di un file JSON: questa scelta è stata dettata dalla semplicità di utilizzo del formato e dalla diensione piuttosto limitata dei dati da memorizzare (il nome dell'utente ed il suo *token* di autenticazione).

La licenza d'uso limitata di Xamarin in nostro possesso ci ha però impedito di utilizzare la stessa libreria JSON su entrambe le piattaforme: abbiamo quindi scelto di aggirare il problema per mezzo di direttive al preprocessore. Useremo cioè i *flag* di compilazione per ignorare alcune righe di codice in base alla piattaforma di destinazione del software.

Il codice compilato su Android userà la libreria System.Json presente nel framework Mono, mentre quello Windows Phone sfrutterà la libreria esterna Newtonsoft.Json, che offre diversi vantaggi sia in termini di efficienza che di facilità di utilizzo. Possiamo vedere un esempio di questa tecnica nel metodo Save(), che salva le proprietà this.username e this.token sul file il cui percorso è contenuto in this._filePath:

```
1 public bool Save()
2 {
3 #if __ANDROID__
4   JsonObject json = new JsonObject();
5   json.Add("username", this.username);
6   json.Add("access_token", this.token == null ? null :
       this.token.access_token);
7   json.Add("token_type", this.token == null ? null :
       this.token.token_type);
```

```
8     json.Add("expires", this.token == null ?
         DateTime.MinValue.ToString() :
         this.token.expires.ToString());
9     string str = json.ToString();
10  #elif SILVERLIGHT
11    var json = new {
12        username = this.username,
13        access_token = this.token == null ? null :
             this.token.access_token,
14        token_type = this.token == null ? null :
             this.token.token_type,
15        expires = this.token == null ?
             DateTime.MinValue.ToString() :
             this.token.expires.ToString()
16    };
17    string str = JsonConvert.SerializeObject(json);
18  #endif
19    try {
20      using (StreamWriter writer = new
           StreamWriter(this._filePath, false)) {
21        writer.Write(str);
22      }
23      return true;
24    }
25    catch (Exception ex) {
26      Console.WriteLine(ex.ToString());
27      return false;
28    }
29  }
```

Business layer: interazione con il web service

La classe `GutenbergClient` fornisce diversi metodi per interagire con il web service. Noteremo che la maggior parte di essi richiede un oggetto `Token`, che verrà utilizzato per autenticare l'utente.

`GetNewToken(string username, string password)` Richiede un nuovo token al web service. Se le credenziali fornite non sono corrette, il metodo lancia un'eccezione; altrimenti deserializza la risposta JSON del server in un oggetto `Token` per poi restituirlo.

`GetIssues(Token t, bool newIssues)` Il valore predefinito di `newIssues` è `false`: in tal caso il metodo recupera la lista degli *issue* acquistati dall'utente.

Se invece si impone `newIssues = true` il metodo restituirà una lista contenente gli *issue* disponibili per l'acquisto.

`BuyIssue(Token t, int id, int magazine_id)` Invia una richiesta per l'acquisto di un *issue*.

`GetPosts(Token t, int id, int magazine_id)` Restituisce una lista degli articoli contenuti in un dato *issue*.

`GetIssueURL(int id, int magazine_id)` Restituisce l'URL di un *issue*.

Le richieste al web service avvengono in modo asincrono e si basano sull'utilizzo di una classe derivata da `System.Net.WebClient`, implementata per rendere personalizzabile il tempo di *timeout* delle connessioni. Questa capacità è per ora limitata ad Android, dato che la libreria *System.Net* di Windows Phone non fornisce ancora un approccio altrettanto intuitivo.

```
1 protected override WebRequest GetWebRequest(Uri address) {
2   WebRequest w = base.GetWebRequest(address);
3   // Default: 10 secondi (Windows Phone)
4   this.timeout = this.timeout == 0 ? 10 * 1000 : this.timeout;
5 #if __ANDROID__
6   w.Timeout = this.timeout; //ms
7 #endif
8   return w;
9 }
```

5.5.2 Interfaccia Android

Panoramica. `Activity` e `Intent`

Lo sviluppo per sistemi Android si basa sul concetto di *activity*. Queste classi descrivono le diverse "schermate" di un'applicazione: forniscono cioè un'interfaccia utente (basandosi su un file XML) e descrivono le operazioni che il dispositivo deve svolgere quando l'utente interagisce con i componenti dell'interfaccia stessa.

Siccome abbiamo previsto un'applicazione dotata di due schermate, realizzeremo altrettante *activity*:

MainActivity Schermata principale: visualizzerà un elenco di *issue* acquistati ed uno di elementi disponibili per l'acquisto.

ViewerActivity Schermata di visualizzazione per il singolo *issue*, che fornirà anche un indice degli articoli in esso contenuti.

Per passare da un'*activity* all'altra useremo oggetti di classe `Intent`. Un `Intent` contiene un riferimento all'*activity* da eseguire ed eventuali dati necessari al suo funzionamento: ad esempio, quello che lancerà la nostra `ViewerActivity` conterrà l'identificativo dell'*issue* da visualizzare.

MainActivity

All'avvio dell'applicazione, l'*activity* principale svolge le seguenti operazioni:

1. Carica l'eventuale `Token` salvato in precedenza;
2. Se il suddetto `Token` non esiste o è scaduto, chiede all'utente di inserire le proprie credenziali in una finestra di dialogo e le usa per ottenere un nuovo token;
3. Ottiene le due liste di *issue* dal web service e le visualizza mediante oggetti `ListView`.

Se l'utente tocca un elemento acquistabile, useremo il `GutenbergClient` creato all'avvio dell'applicazione per inviare una richiesta d'acquisto: in caso di esito positivo, l'*activity* si occuperà di spostare l'elemento nella lista degli *issue* in possesso dell'utente.

Se invece viene selezionato un elemento già acquistato, faremo partire la `ViewerActivity` per mezzo di un `Intent`.

```
1 issuesList.ItemClick += delegate(object sender,
     AdapterView.ItemClickEventArgs args) {
2   Intent i = new Intent(this, typeof(ViewerActivity));
3   i.PutExtra("id", adapter[args.Position].id);
4   i.PutExtra("magazine_id", adapter[args.Position].magazine_id);
5   this.StartActivity(i);
6 };
```

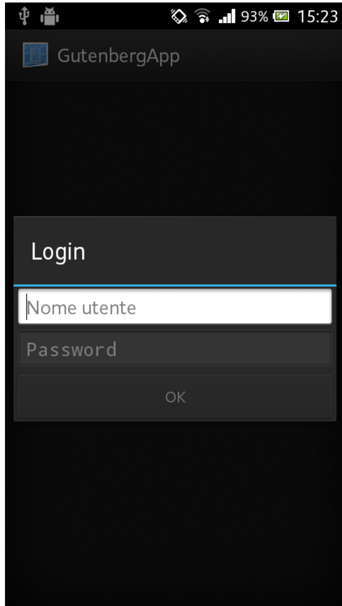


Figura 5.3: Schermata principale dell'applicazione Android. Se l'utente non ha un token valido, viene visualizzata una finestra di login.

ViewerActivity

Questa *activity* è costituita principalmente da una WebView, cioè un elemento per la visualizzazione di pagine web. In fase di avvio, l'*activity* recupera i dati contenuti nell'Intent da cui è stata lanciata e li usa per visualizzare la prima pagina dell'*issue*.

```
1  this.prefs = new Preferences();
2  this.id = Intent.GetIntExtra("id", -1);
3  this.magazine_id = Intent.GetIntExtra("magazine_id", -1);
4  this.client = new GutenbergClient();
5  this.url = this.client.GetIssueUrl(this.id, this.magazine_id);
6  this.posts = await this.client.GetPosts(this.prefs.token,
       this.id, this.magazine_id);
7  // Aggiungo il token all'header che verra' usato dalla WebView
8  var headers = new Dictionary<string, string>();
9  headers.Add("Authorization", this.prefs.token.token_type + " " +
       this.prefs.token.access_token);
10 this.webview.LoadUrl(this.url+"/index.html", headers);
```

L'oggetto `webview` è di una classe che estende la `WebView` predefinita di Google in modo da associare il token dell'utente a tutte le richieste emesse durante la visualizzazione.

La consultazione dell'*issue* può avvenire in due modi: scorrendo singolarmente le pagine oppure selezionando uno degli articoli da un menu che appare quando l'utente schiaccia l'omonimo pulsante del proprio dispositivo. In quest'ultimo caso, useremo un comando JavaScript per indicare a Treesaver quale articolo dovrà essere visualizzato.

```
1 menuList.ItemClick += delegate(object sender,
     AdapterView.ItemClickEventArgs args) {
2   this.webview.LoadUrl(String.Format(
3     "javascript:treesaver.goToDocumentByURL('{0}')",
4     this.url + "/" + adapter[args.Position].url));
5   menuDialog.Dismiss();
6 };
```

5.5.3 Windows Phone

Panoramica

Le applicazioni Windows Phone seguono un paradigma leggermente diverso:

- Ciascuna schermata (*pagina*) è descritta da un file XAML[1];
- Il nodo radice del documento XAML contiene un riferimento ad una classe C#;
- I gestori degli eventi saranno metodi della suddetta classe, che verranno associati agli elementi dell'interfaccia aggiungendo una proprietà `evento="nome_metodo"` al relativo nodo nel documento XAML;
- Il passaggio da una pagina ad un'altra avviene grazie alla classe `NavigationService`, che si baserà sulle URL dei file XAML. Sarà possibile passare dati alla nuova pagina sotto forma di query-string oppure raccogliendoli in un oggetto.

```
1 NavigationService.Navigate("Page.xaml");
2 NavigationService.Navigate("Page.xaml?id=1");
```

[1] XAML (*Extensible Application Markup Language*) è un linguaggio dichiarativo basato su XML e sviluppato da Microsoft.

```
3
4 var data = new { id = 1 };
5 NavigationService.Navigate("Page.xaml", data);
```

Abbiamo realizzato tre pagine, due delle quali sono analoghe alle *activity* dell'applicazione Android; la terza è una pagina di login.

Main.xaml

La pagina principale svolge le stesse funzioni del proprio omologo in Android. Il seguente metodo svolge le stesse funzionalità del blocco di codice visto nella sezione dedicata a `MainActivity`: lo riportiamo per rendere evidenti le differenze concettuali alla base dei due sistemi operativi.

```
1  private void IssuesListBox_SelectionChanged(object sender,
     SelectionChangedEventArgs e) {
2    if (IssuesListBox.SelectedIndex != -1) {
3      var item = (Issue)IssuesListBox.SelectedItem;
4      NavigationService.Navigate(new Uri(
5        string.Format(
6          "/ViewerPage.xaml?id={0}&magazine_id={1}",
7          item.id,
8          item.magazine_id),
9        UriKind.Relative
10      ));
11     IssuesListBox.SelectedIndex = -1;
12   }
13 }
```

Le differenze nell'interfaccia grafica (figura 5.5.3) sono evidenti: abbiamo cercato di adattarci ai diversi "canoni" di ciascun sistema operativo.

`Login.xaml`

La pagina `Main.xaml` chiamerà la schermata di login se l'utente è sprovvisto di un *token* valido. Quest'ultima si limiterà a raccogliere i dati forniti dall'utente e ad inviarli alla pagina principale, che li userà per ottenere un nuovo token (grazie ad un oggetto `GutenbergClient`).

```
1 private void btnLogin_Click(object sender, RoutedEventArgs e) {
2   NavigationService.Navigate(new Uri(
```

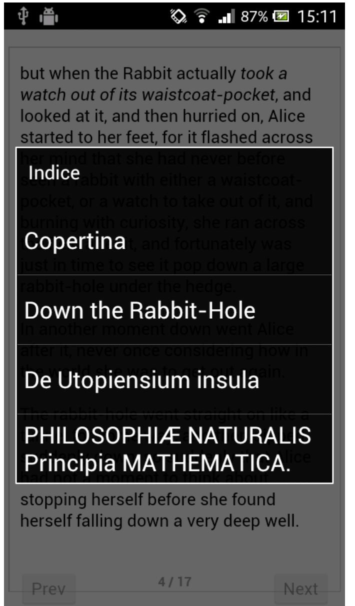
```
3        string.Format(
4          "/Main.xaml?username={0}&password={1}",
5          txt_username.Text,
6          txt_password.Password),
7        UriKind.Relative
8      ));
9    }
```

Figura 5.4: Schermata principale dell'applicazione Windows Phone: è possibile passare dalla pagina "Riviste" a quella "Nuove uscite" scorrendo il dito sullo schermo. La terza immagine mostra la schermata di login.

`ViewerPage.xaml`

Anche questa pagina non presenta sostanziali differenze rispetto all'*activity* di Android: useremo un elemento `WebBrowser` per visualizzare le pagine che costituiscono l'*issue* scelto dall'utente.

Siccome i dispositivi Windows Phone sono sprovvisti di un apposito pulsante, il menu degli articoli viene richiamato facendo scorrere il dito su un elemento `ApplicationBar` sempre presente nella parte inferiore dello schermo.



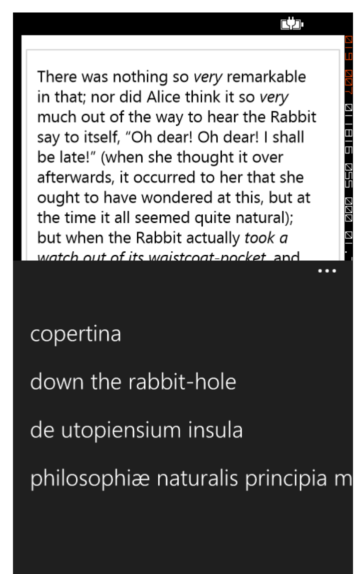


Figura 5.5: Differenze tra l'interfaccia dell'applicazione Android e quella realizzata su Windows Phone.

Limitazioni e sviluppi futuri

Durante lo sviluppo abbiamo riscontrato alcuni limiti che rendono l'applicazione Windows Phone meno "matura" rispetto alla versione per Android. In particolare, il componente `WebBrowser` non consente la personalizzazione degli *header* HTTP, impedendoci di applicare automaticamente il token a tutte le richieste in uscita: ciò significa che, allo stato attuale di sviluppo, l'applicazione può accedere agli *issue* solo se sul web service è stata disabilitata l'autenticazione per file statici.

A meno che l'API del componente non venga migliorata nelle prossime versioni di Windows Phone, una soluzione potrebbe essere quella di modificare il web service in modo che il *token* possa essere allegato tramite *cookie*.

Conclusioni e sviluppi futuri

Gli obiettivi che ci eravamo prefissati sono stati raggiunti in modo soddisfacente: abbiamo dimostrato la possibilità di gestire l'intero ciclo di vita dei prodotti editoriali attraverso un numero limitato di componenti, basati in larga maggioranza su tecnologie comuni come il framework .NET ed il linguaggio C#, facilmente scalabili e caratterizzati da costi di sviluppo e manutenzione ridotti.

Il progetto appena descritto, tuttavia, resta di carattere prettamente sperimentale. Per arrivare ad un prodotto commercializzabile sarà necessario affrontare ulteriori fasi di sviluppo:

Sviluppo su iOS Fermo restando che le componenti di base delle applicazioni *mobile* saranno direttamente riutilizzabili anche sul sistema Apple, basterà riscrivere il *presentation layer* per rendere pressoché completa la gamma dei dispositivi supportati dalla nostra applicazione.

***Deployment* automatizzato** Allo stato attuale, i singoli componenti devono essere inizializzati attraverso la modifica dei rispettivi file di configurazione. Ad esempio, sia il back-end che il web service devono conoscere le credenziali d'accesso al DBMS; il front-end deve sapere l'indirizzo del servizio WPF di back-end; eccetera. Per rendere più chiaro e veloce il *deployment* bisognerà realizzare un software capace di automatizzare il processo di configurazione della piattaforma.

Connessioni sicure al web service I messaggi diretti al web service dovranno necessariamente avvenire attraverso connessioni sicure, visto che possono contenere un `token` di autenticazione o le credenziali stesse dell'utente.

Supporto ai pagamenti elettronici L'utente potrebbe avere accesso ad un *issue* solo dopo aver effettuato una qualche forma di pagamento: anche in questo caso sarà indispensabile il ricorso a sistemi di comunicazione sicuri, come quelli forniti dal protocollo HTTPS.

Bibliografia

[1] AGCOM. Osservatorio trimestrale telecomunicazioni - aggiornamento al 30 settembre 2013, 2013.
http://www.agcom.it/default.aspx?DocID=12264.

[2] Cisco Systems, Inc. Cisco visual networking index: Global mobile data traffic forecast update, 2013–2018, 2014.
http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[3] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, The Internet Group, 2006.
http://tools.ietf.org/html/rfc4627.

[4] ECMA-334. C# Language Specification. 4th Edition, 2006.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[5] ECMA-335. Common Language Infrastructure (CLI). 6th Edition, 2012.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf.

[6] The WordPress Foundation. Wordpress, 2003-2013.
http://www.wordpress.org.

[7] The PHP Group. Original MySQL API. Introduction, 2013.
http://www.php.net/manual/en/intro.mysql.php.

[8] IDC Italia Srl. Tablet e smartphone trainano la domanda di smart connected device in Italia. Comunicato stampa, Dicembre 2013.

http://idcitalia.com/dwn/SF_103294/cs_tablet_e_smartphone_trainano_la_domanda_in_italia_final.pdf.

[9] ISO/IEC 23270:2006. Information technology – Programming languages – C#, 2006.
http://standards.iso.org/ittf/PubliclyAvailableStandards/c042926_ISO_IEC_23270_2006(E).zip.

[10] ISO/IEC 23271:2012. Information technology – Common Language Infrastructure (CLI). Third Edition, 2012.
http://standards.iso.org/ittf/PubliclyAvailableStandards/c058046_ISO_IEC_23271_2012(E).zip.

[11] Michael Jones, Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, The Internet Group, 2012.
http://tools.ietf.org/html/rfc6750.

[12] Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide*. Microsoft Press, 2009.
http://msdn.microsoft.com/en-us/library/ff650706.aspx.

[13] Il Sole 24 Ore. Banche Dati Professionali, 2014.
http://www.banchedati.ilsole24ore.com/.

[14] Roger Black Studio, Inc. Treesaver.js, 2011-2013.
http://treesaverjs.com.

[15] Xamarin. Mono, 2004-2014.
http://www.mono-project.com/.