ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Scienze di Internet

Sviluppo di componenti per una piattaforma web database-independent

Tesi di laurea in
Tecnologie Web

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Marco Scaruffi

Anno Accademico 2012/2013 • Sessione III

Indice

Capitolo 1 – Introduzione

Lo scopo di questo lavoro è realizzare, in ambito web, un'applicazione client-server *database-independent*, ovvero un'applicazione il cui funzionamento non è vincolato da uno specifico tipo di base di dati. Ci sono diversi motivi per cui è desiderabile sviluppare un'applicazione che presenta tale caratteristica.

Anzitutto, nell'epoca del web 2.0, dei social network, del cloud computing, in un periodo in cui il mercato delle basi di dati è in fermento e sorgono nuove tecnologie di continuo, chi sviluppa un'applicazione massicciamente dipendente da uno specifico tipo di base di dati rischia di ritrovarsi in breve tempo con un prodotto obsoleto, di dover riscrivere ampie parti di codice per adattarlo ad una nuova tecnologia, o di dover rinunciare al passaggio ad una tecnologia per mancanza di tempo e risorse.

Inoltre, a prescindere dal fatto che in futuro si abbia intenzione o meno di passare ad un'altra tecnologia, un'applicazione web *database-independent* risulta più facile da gestire riguardo ad aspetti come sicurezza, scalabilità e manutenzione.

Intorno al 2000, Roy Thomas Felding (studente della University of California) presenta nella sua tesi il modello *REST* (acronimo di *REpresentational State Transfer*), ovvero una serie di linee guida implementative utili a sviluppare un'applicazione web client-server definibile come *database-independent.* Queste linee guida verranno analizzate meglio nel prossimo capitolo, ma sinteticamente può essere definita *RESTful* un'applicazione client-server che effettua transazioni in modo che ogni parte coinvolta comunichi con le altre senza conoscerne la struttura, ma utilizzando uno standard (ad esempio *XML* o *JSON*). [RTF00]

Sul mercato attualmente sono disponibili alcune applicazioni e *API RESTful* (come ad esempio *SlashDB*) che permettono di fare da tramite tra una o più parti client e una o più basi di dati. Il limite principale di queste applicazioni (almeno per il momento) è il fatto di supportare un range limitato di tecnologie, che al momento non include basi di dati che trascendano il modello classico relazionale.

L'applicazione presentata in questa trattazione si prefigge l'obiettivo di superare questo limite.

FimoDB è un'applicazione web *RESTful database-independent* composta da un lato client scritto in *Javascript, HTML, CSS* e un corrispettivo lato server scritto in *PHP* e *XML*.

La potenza di questa applicazione sta, per quanto riguarda il lato client, nella genericità del suo utilizzo, al punto che avrebbe quasi più senso parlare di un'applicazione a se stante: a patto di seguire una serie di linee guida, possono essere sviluppati sistemi gestionali web di qualsiasi tipo.

Tuttavia, questa trattazione sarà incentrata principalmente sul lato server e sui suoi componenti. In particolare verranno descritti i moduli di connessione alle varie basi di dati.

Il lato server è composto da una serie di files *PHP* (contenenti classi, funzioni e procedure) e da una serie di files di configurazione *XML* (da cui vengono prelevate informazioni). Brevemente, il cuore del lato server è costituito da un file procedurale detto *dispatcher,* che ha la funzione di istanziare i moduli di connessione alle basi di dati, e di passare a questi le informazioni per effettuare le operazioni.

La parte di codice che si occupa di effettuare operazioni sulle basi di dati prende il nome di modulo di connessione o pillola. Ogni tipo di base di dati ha un modulo relativo (ad esempio per *Mysql* esiste un modulo chiamato *Mysql*) contenuto in un apposito file *PHP*. Questo file contiene una classe i cui oggetti vengono istanziati dal *dispatcher*. Questa classe contiene le funzioni relative alle operazioni *CRUD* effettuabili su una base di dati. Verrà mostrato nel capitolo 2 che queste operazioni, sebbene differiscano nei modi da tecnologia a tecnologia, sono sempre le stesse. Una volta effettuata la richiesta alla base di dati, la funzione del modulo di connessione decodifica la risposta e la converte in formato *XML*, in modo che possa essere letta dal *dispatcher*.

Ad ogni modulo è associato un file di configurazione scritto in *XML*. Questo file contiene le informazioni necessarie al modulo per portare a termine una operazione sulla base di dati. Ad ogni richiesta il *dispatcher* legge il file e passa gli argomenti all'oggetto del modulo istanziato.

La forza di questo approccio sta nel fatto che, essendo i moduli dislocati dal resto del codice, per aggiungere il supporto ad una nuova tecnologia è sufficiente creare un nuovo modulo, senza che il

codice già esistente venga intaccato.

Le difficoltà principali incontrate durante lo sviluppo di FimoDB derivano dal fatto che, sebbene le operazioni effettuate siano le stesse per ogni base di dati, la struttura di quest'ultima cambia da tecnologia a tecnologia. Succede quindi che un'operazione molto semplice da eseguire su una tecnologia si riveli estremamente complicata su un'altra. E' il caso, ad esempio, delle tecnologie *NoSQL*, ma anche delle basi di dati classiche, ancorate al modello relazionale. In tutti i casi è comunque stato possibile trovare una soluzione per effettuare le operazioni desiderate.

Le migliorie che si potrebbero apportare a FimoDB (allo stato attuale) sono le seguenti:

- trovare una soluzione più efficiente/usabile per alcune procedure (o alcuni algoritmi).
- estendere il supporto a nuove tecnologie implementando i relativi moduli.
- aggiungere funzionalità (popolare funzioni vuote/crearne di nuove).

Capitolo 2 – Contesto scientifico e tecnologico

In questo capitolo verrà fornita una panoramica del contesto inerente al problema di questa trattazione: lo sviluppo di un'applicazione web *database-independent*. Verranno analizzati:

- Struttura e limiti di una piattaforma che dipende in modo vincolante dal tipo di database utilizzato.
- Soluzioni generiche per superare il suddetto vincolo.
- Il modello REST.
- Soluzioni (gratuite e a pagamento) che implementano il modello REST.

2.1 Panoramica e limiti degli approcci database-dependent

Si può definire *database-dependent* una piattaforma o applicazione progettata per funzionare, nella sua interezza, esclusivamente con un determinato tipo di base di dati (dove per tipo si intende una tecnologia specifica, non un modello generico). Il codice di un'applicazione *database-dependent* può assumere le sfaccettature più disparate, ma quello che interessa in questa sede è il fatto che la parte di codice che si occupa di effettuare le operazioni sulla base di dati sia inscindibile dalle altre parti, e tra queste parti non sia ben definita una linea di demarcazione. Va precisato che un'applicazione può anche supportare tecnologie differenti, ma se all'aggiunta del supporto per una particolare tecnologia deve essere riscritta completamente (o in gran parte), rimane comunque un'applicazione vincolata.

Esempi popolari di applicazioni web *database-dependent* possono essere *Content-Management-System* come *WordPress* [Wor13a], *Joomla* [Joo13], *Drupal* [HH13], *Magento* [Mag13], *Oscommerce* [Osc13]. Sebbene alcune versioni di queste applicazioni supportino l'installazione su più tipi di basi di dati, in nessun caso può essere integrata una nuova tecnologia senza dover riscrivere grandi parti di codice e/o limitare funzionalità.

Analizzando nel dettaglio la struttura di una piattaforma *database-dependent* come *WordPress* (che supporta ufficialmente solo *Mysql*), risulta evidente che non sia definita una parte di codice adibita esclusivamente alle operazioni su basi di dati: ogni *plugin* installato effettua operazioni indipendentemente dalla parte *core* della piattaforma. [Wor13b]
Questo vincolo diventa un problema nel momento in cui chi amministra una piattaforma, o chi sviluppa il codice di un'applicazione, ha la necessità di utilizzare come base di dati una tecnologia correntemente non supportata.

I problemi in cui si rischia di incorrere sono i seguenti:

- Dover rinunciare (in parte o totalmente) ad alcune features per mancanza di compatibilità.
- Dover riprogettare (in parte o totalmente) l'applicazione, perdendo tempo e risorse.
- Sincronizzare ogni versione della piattaforma (relativa ad una determinata tecnologia) ad ogni nuovo aggiornamento.

La conseguenza di questi problemi è il rischio di incappare nell'effetto *lock-in* (di tipo tecnologico) qualora il costo del passaggio ad una nuova tecnologia, in termini economici o di tempo, fosse maggiore del vantaggio ottenuto dall'utilizzo di tale tecnologia. [SV99]
Più lo stadio di sviluppo di un'applicazione *database-dependent* è evoluto, più diventa problematico passare ad una nuova tecnologia. Nel caso di un'applicazione come *Wordpress* (che vanta più di 70 milioni di installazioni [Wor13c]) nonostante le numerose richieste da parte della community, gli sviluppatori non hanno mai preso in considerazione l'idea di supportare ufficialmente più tecnologie di basi di dati, principalmente perché, per le motivazioni sopra elencate, sarebbe troppo complicato [Wor13b].

2.2 Approcci database-independent

Una piattaforma *database-independent* è progettata per funzionare indipendentemente dal tipo e dalla struttura della base di dati utilizzata. La parte di codice che si occupa di effettuare operazioni sulla base di dati è separata dal resto. Generalmente è strutturata a moduli facilmente espandibili. E' quindi una soluzione adatta per chi ha bisogno di passare in modo agevole da una tecnologia all'altra in modo arbitrario.

2.2.1 Esempio di implementazione database-independent

Arnab Roy Chowdhury [Cho12] propone un esempio di applicazione *database-independent* realizzata in *C#*, strutturata come segue:

- Un tipo di classe astratta relativa al consumer (non si può parlare di client perché non è un'implementazione client-server).
- Un tipo di classe astratta generica che gestisce le operazioni sulle basi di dati.
- Classi relative ad specifica tecnologia che implementano le suddette classi. Siccome la parte consumer si interfaccia alle classi astratte, non c'è bisogno di avere informazioni su come le funzioni di queste classi sono state implementate.
- Un file di configurazione scritto in *XML*, che, per ogni tecnologia, contiene i parametri di connessione alla relativa base di dati.

Sebbene a livello concettuale la struttura sia corretta, uno sguardo all'implementazione del codice rende palese una serie di limiti:

- L'applicazione può funzionare solo con basi di dati di tipo relazionale.
- L'implementazione da per scontato che il linguaggio SQL sia universale.
- L'implementazione è vincolata da una specifica struttura della base di dati. All'interno delle funzioni compaiono esplicitamente nomi di tabelle.

Tra qualche paragrafo verrà mostrato come è possibile aggirare questi limiti, adottando una serie di linee-guida implementative.

2.2.2 Critiche agli approcci database-independent

Le principali critiche rivolte agli approcci *database-independent* sono le seguenti [AT12]:

- Non può esistere un'applicazione veramente *database-independent*, in quanto il codice viene testato su un numero limitato di versioni della stessa tecnologia e non è garantito che

funzioni in caso di versioni future.

- I tempi di sviluppo sono lunghi: per ogni nuova tecnologia bisogna riscrivere tutta la parte di codice relativa alle operazioni sulla base di dati.
- Non è detto che, a livello concettuale, un'applicazione possa funzionare su qualsiasi database.
- Le performances delle applicazioni *database-independent* sono scadenti.
- Gli sviluppatori devono avere una conoscenza specifica di ogni nuova tecnologia che implementano, e a livelli avanzati non è sempre possibile.

Queste critiche sono in parte condivisibili, ma poggiano su presupposti vaghi e aleatori. Anzitutto non vengono mosse nei confronti di applicazioni reali, ma di uno stereotipo di applicazione non meglio precisato. Le applicazioni *database-independent* non sono una macchia uniforme: hanno tante sfaccettature diverse. Dipende quindi tutto da come vengono concepite e implementate.

E' vero che non c'è garanzia del fatto che un'applicazione testata su un numero limitato di versioni della stessa tecnologia funzioni anche con versioni future, ma se a livello concettuale l'applicazione è solida, aumentare il range di compatibilità sarà semplice e veloce. Va poi aggiunto che è rarissimo che una base di dati venga stravolta da una versione alla successiva.

I tempi di sviluppo sono più lunghi rispetto a quelli di un'applicazione *database-dependent*, ammesso che chi abbia sviluppato quest'ultima non scopra in futuro di avere bisogno di migrare ad una nuova tecnologia. Inoltre, se chi sviluppa un'applicazione *database-independent* in fase di implementazione stabilisce di supportare momentaneamente solo una tecnologia, i tempi di sviluppo sono gli stessi (se non inferiori) previsti per un'applicazione *database dependent.*

Le operazioni che un'applicazione effettua su una base di dati sono limitate (verranno descritte meglio nel prossimo paragrafo), e supportate da qualsiasi tecnologia nell'ambito delle basi di dati.

Le performance e il livello di conoscenza della tecnologia dipendono da tempi e risorse degli sviluppatori. E' difficile che un solo programmatore abbia conoscenze tali da definirsi esperto di tutte le tecnologie presenti sul mercato, ma se il progetto è sviluppato da una community, chi è esperto di una particolare tecnologia può dedicarsi a sviluppare solo la parte di codice inerente, ed

una volta messa insieme alle altre, si otterrà un'applicazione completa e performante.

Concludendo, non esiste una soluzione di sviluppo intrinsecamente migliore. Dipende dai casi. A volte non è neanche possibile scegliere che soluzione adoperare (magari non si hanno conoscenze e competenze). Certo è che, nel 2014, il mercato delle basi di dati è dinamico e in fermento e se non ci si vuole ritrovare con soluzioni obsolete (e rimanere tagliati fuori), è necessario stare al passo.

2.3 Operazioni effettuabili su una base di dati: la tavola CRUD

Qualsiasi richiesta effettuata nei confronti di una base di dati, indipendentemente dalla tecnologia utilizzata, può essere ricondotta ad un elemento dell'insieme di operazioni meglio noto come *CRUD* (acronimo di *Create, Read, Update, Delete*). La condizione fondamentale per effettuare queste operazioni è l'essere in possesso dei permessi necessari [WP14]. Ogni tecnologia possiede equivalenti di queste operazioni. Di seguito alcuni esempi:

Operazione	Equivalente *SQL* (tutti)	Equivalente *MongoDB*	Equivalente *BaseX(xQuery)* [Bas14]
Create	INSERT	*insert()*	insert
Read	SELECT	*find()*	doc(risorsa);
Update	UPDATE	*update()*	replace
Delete	DELETE	*remove()*	delete

In un'applicazione *database-independent* è sempre possibile implementare queste operazioni, a prescindere dalla tecnologia utilizzata. Va precisato però che queste sono operazioni base e non è detto che richieste più specifiche possano essere realizzate su tutte le tecnologie.

2.4 Il modello REST

Uno dei paradigmi (o stili architetturali) che possono essere seguiti per sviluppare un'applicazione

database-independent è il cosiddetto modello *REST,* acronimo di *Representational State Transfer*. Questo modello delinea una serie di vincoli che vanno seguiti per mantenere l'indipendenza della base di dati (e in generale della parte server) dalla parte client dell'applicazione. Si definisce *RESTful* un'applicazione client-server che effettua operazioni di tipo *CRUD* (descritte nel paragrafo precedente) tramite protocollo *HTTP,* rispettando i seguenti vincoli [RTF00]:

- *Uniformità dell'interfaccia:* durante ogni transazione, il server invia al client contenuti in formato standard (tipicamente *JSON* o *XML*). Per effettuare un'operazione su una determinata risorsa, il client deve avere i permessi e le informazioni necessarie. Ogni transazione deve essere indipendente dalle altre.
- *Assenza di stato (sessione):* tutte le informazioni necessarie per gestire una richiesta di qualsiasi tipo sono contenute nella richiesta stessa. Di conseguenza non viene stabilita una sessione durante la quale vengono inviate richieste multiple, ma ogni richiesta è una sessione in sé.
- *Possibilità di salvare i risultati in cache:* tutte le transazioni possono essere salvate nella *cache* del client, che evita così di effettuare più volte la stessa richiesta, ottimizzando le performances.
- *Client e server indipendenti:* Il client e il server vengono sviluppati in modo indipendente e intercambiabile.
- *Struttura a livelli:* Il client non può mai determinare se è connesso al server che ospita la base di dati, o se è connesso ad un server intermediario.

Un'applicazione che rispetta questi vincoli risulterà semplice, scalabile, portabile, affidabile e facilmente espandibile.

2.5 Esempi di applicazioni *RESTful Database-Independent*

Verrà ora fornita una panoramica delle due principali soluzioni *RESTful database-independent* (commerciali e gratuite) attualmente disponibili sul mercato. A grandi linee il funzionamento queste applicazioni è lo stesso: un'interfaccia tra la parte client dell'applicazione e la base di dati, che

effettua operazioni di tipo *CRUD* sulla seconda e restituisce il risultato alla prima. Non verrà quindi analizzata in dettaglio la struttura implementativa, ma verranno fornite informazioni su punti di forza e limiti di ogni applicazione.

2.5.1 SlashDB

SlashDB è un'applicazione a pagamento che permette di interrogare una base di dati e convertire in modo automatico il risultato in un formato universale (XML, JSON, CSV) leggibile dal client. [Sla14a]

L'applicazione può funzionare in due modi:

- *Data discovery:* tutto il contenuto del database viene estratto e reso consultabile in formato HTML.
- *SQL Pass-thru:* l'utente crea una query specifica che viene eseguita dall'applicazione.

I punti di forza di *SlashDB* sono:

- Facilità di utilizzo (interfaccia grafica per impostare i parametri).
- Supporta il *Cloud Computing*.

I limiti di *SlashDB* sono:

- Supporta solamente le basi di dati di tipo relazionale (anche se in futuro c'è la possibilità che venga aggiunto il supporto per *MongoDB* [Sla14b]).
- Bisogna installare una macchina virtuale.
- Non è un'applicazione *open-source* (requisito base per dare vita ad una community che sviluppi moduli aggiuntivi).
- L'applicazione si limita a convertire il risultato di una richiesta in formato leggibile dal client, senza effettuare ulteriori modifiche.

2.5.2 RESTSQL

RESTSQL è un'applicazione *open-source* scritta in *Java* che permette di interrogare una base di dati e convertire il risultato in formato XML. [RS14]

L'applicazione può funzionare in due modi:

- Interfaccia HTTP client-server.
- Api Java

I punti di forza di RESTSQL sono:

- Query automatizzate.
- Applicazione open-source (può nascere una community che sviluppi moduli aggiuntivi).

RESTSQL al momento presenta i seguenti limiti:

- Sono supportate solamente le basi di dati di tipo relazionale.
- L'applicazione si limita a convertire il risultato di una richiesta in formato leggibile dal client, senza effettuare ulteriori modifiche.
- Potenza delle query limitata.

2.6 Supporto di basi di dati non relazionali

Da uno sguardo alla panoramica appena fornita, risulta che il limite principale delle applicazioni attualmente disponibili sul mercato è il supporto di basi di dati esclusivamente di tipo relazionale. La causa di questo limite non è di tipo tecnico: è stato mostrato nei paragrafi precedenti che un'applicazione *RESTful* è facilmente espandibile, e che qualsiasi tecnologia supporta operazioni di tipo *CRUD*. Verrà inoltre mostrato nei capitoli successivi che sviluppare moduli per basi di dati non relazionali (*NoSQL*, *XML*, ecc..) risulta più facile rispetto allo sviuppo per basi di dati di tipo

relazionale.

Il motivo principale della mancanza di supporto relativo a basi di dati non relazionali è che nell'attuale ranking relativo alle tecnologie più utilizzate (riguardo alle basi di dati), le prime dieci risultano essere tutte basi di dati di tipo relazionale, con l'eccezione di *MongoDB* e *Cassandra*. Tuttavia, tra le prime tre tecnologie in classifica (*Oracle*, *Mysql*, *Microsoft Sql Server*) e le successive, risulta un rapporto di utilizzo di circa 10 a 1. [Dbe14]

Gli sviluppatori hanno quindi preferito concentrarsi sulle tecnologie più utilizzate, per asservire alle logiche di mercato. Come detto in precedenza però, il mercato è dinamico e le alternative diventano sempre più popolari (basti pensare ad esempio che tutti i principali social network utilizzano tecnologie di tipo *NoSQL*).

Nei prossimi capitoli verrà presentata nel dettaglio un'applicazione web *RESTful* che supporta anche basi di dati di tipo non relazionale.

Capitolo 3 – FimoDB: una panoramica ad alto livello

Questo capitolo fornisce una panoramica ad alto livello del progetto FimoDB. Verranno illustrate:

- Struttura e funzionalità dell'applicazione (lato client e lato server).
- Migliorie rispetto alle soluzioni attualmente presenti sul mercato.

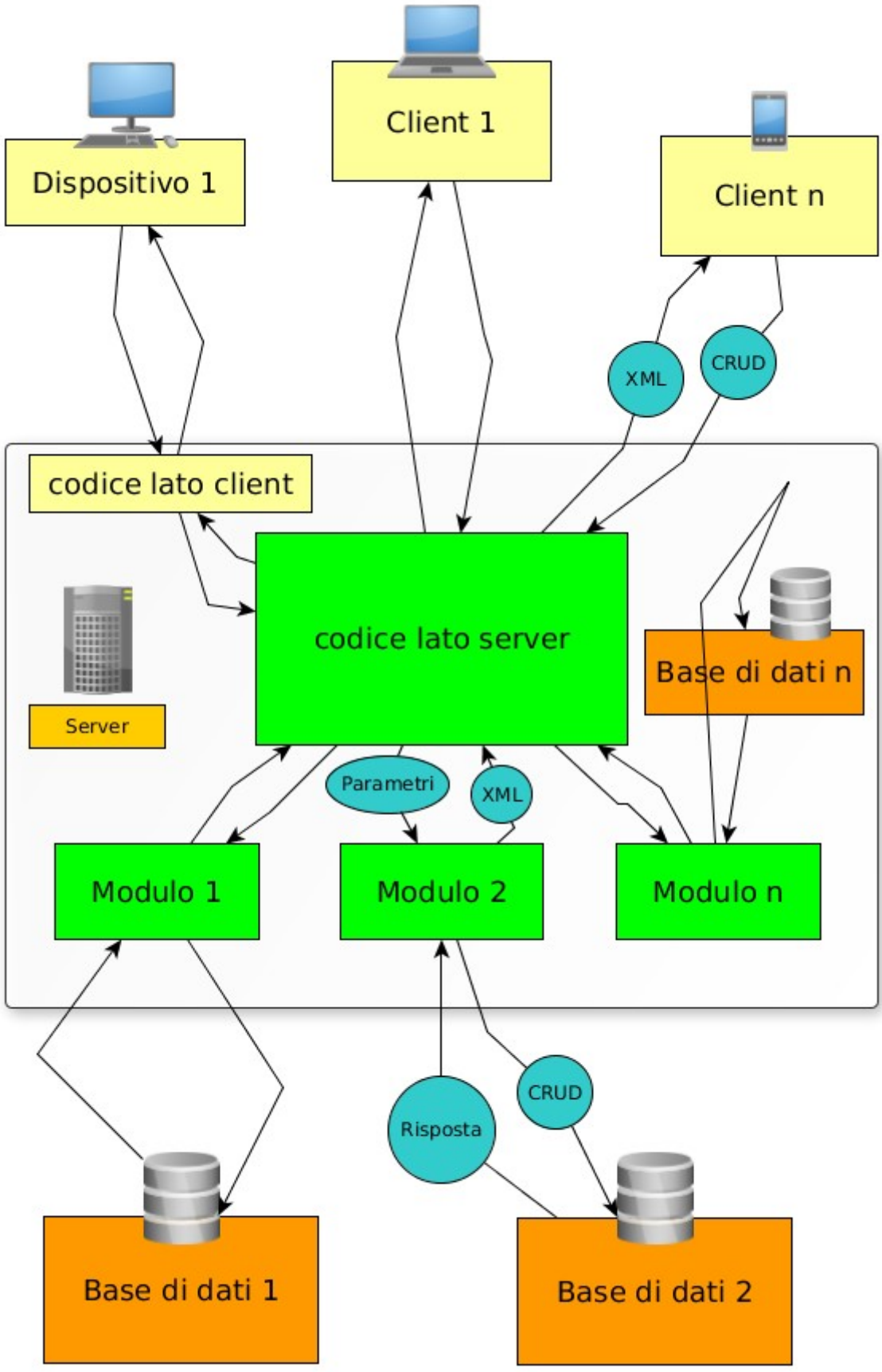
Verranno inoltre forniti esempi di utilizzo dell'applicazione e relativi casi d'uso.

3.1 Il progetto FimoDB

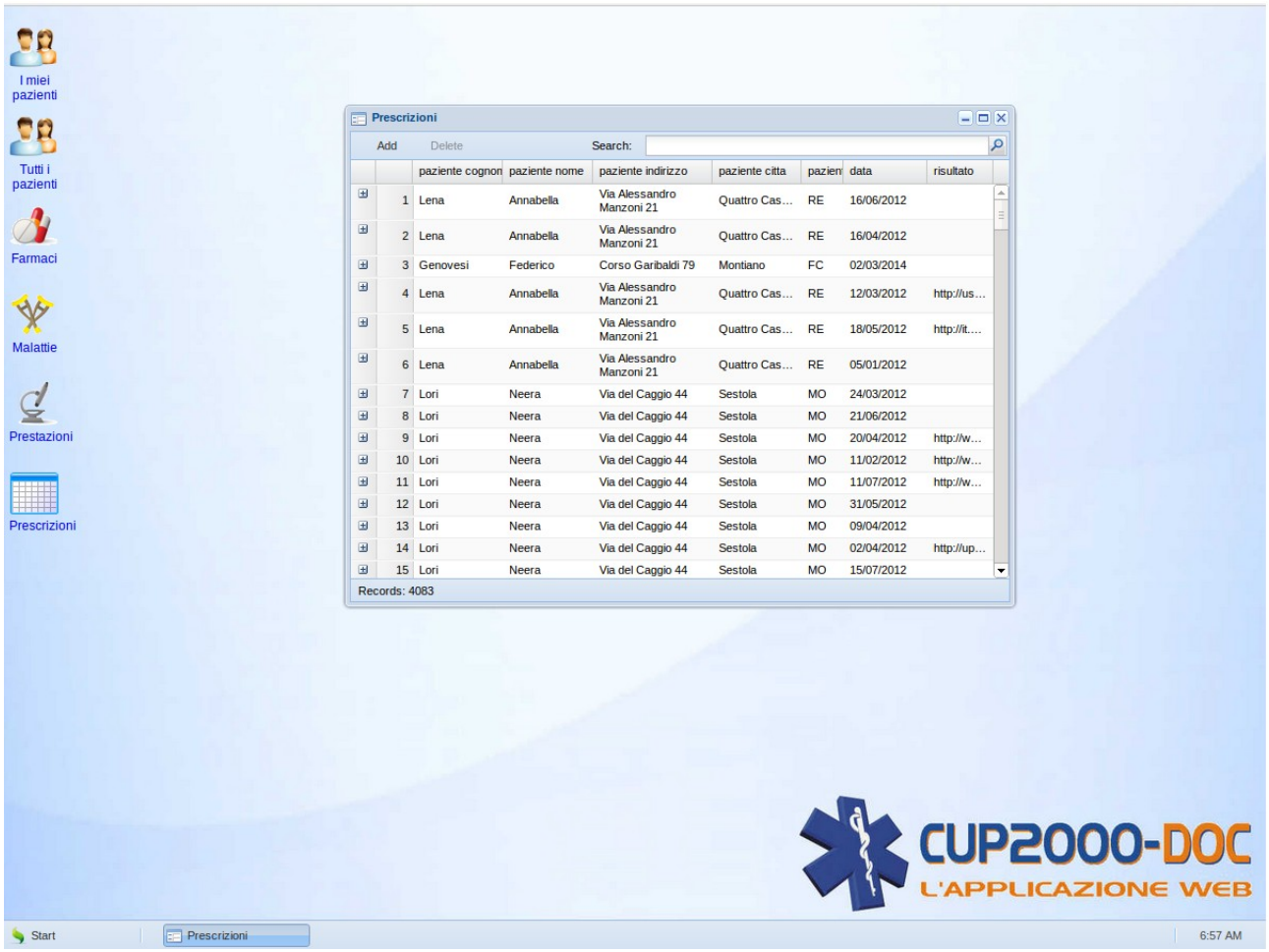
FimoDB è un'applicazione *RESTful database-independent* che permette al lato client (sviluppato seguendo una serie di linee guida) di inviare richieste HTTP di tipo *CRUD* al lato server (per dettagli su *REST* e *CRUD* si rimanda al capitolo 2). L'immagine [FIG01] fornisce una panoramica generale della struttura dell'applicazione e del flusso di lavoro.

3.1.1 Il lato client

Seguendo le logiche del modello *REST*, sarebbe più corretto definire il lato client di FimoDB come un'applicazione a se stante. Utilizzando *Ext Js*, un framework *Javascript* che, tramite *Ajax*, *Jquery* e altre tecnologie, permette di creare applicazioni web potenti, dinamiche e dall'aspetto gradevole, le possibilità di sviluppo sono pressoché illimitate: sistemi gestionali di vario tipo, e-commerce, piattaforme di publishing, ecc... L'applicazione attiva al momento è un sistema di gestione di ricette mediche. [FIG02]
Riguardo all'ubicazione del codice, questo può trovarsi sul server che contiene il codice lato server, su un server intermediario, oppure direttamente sulla macchina client.
Per interagire con il server, il client invia una richiesta di tipo *CRUD* che contiene parametri arbitrari. Il server elabora la richiesta (verrà spiegato come nel prossimo paragrafo) e restituisce la risposta in formato XML.



[FIG01] Flusso di lavoro dell'applicazione.

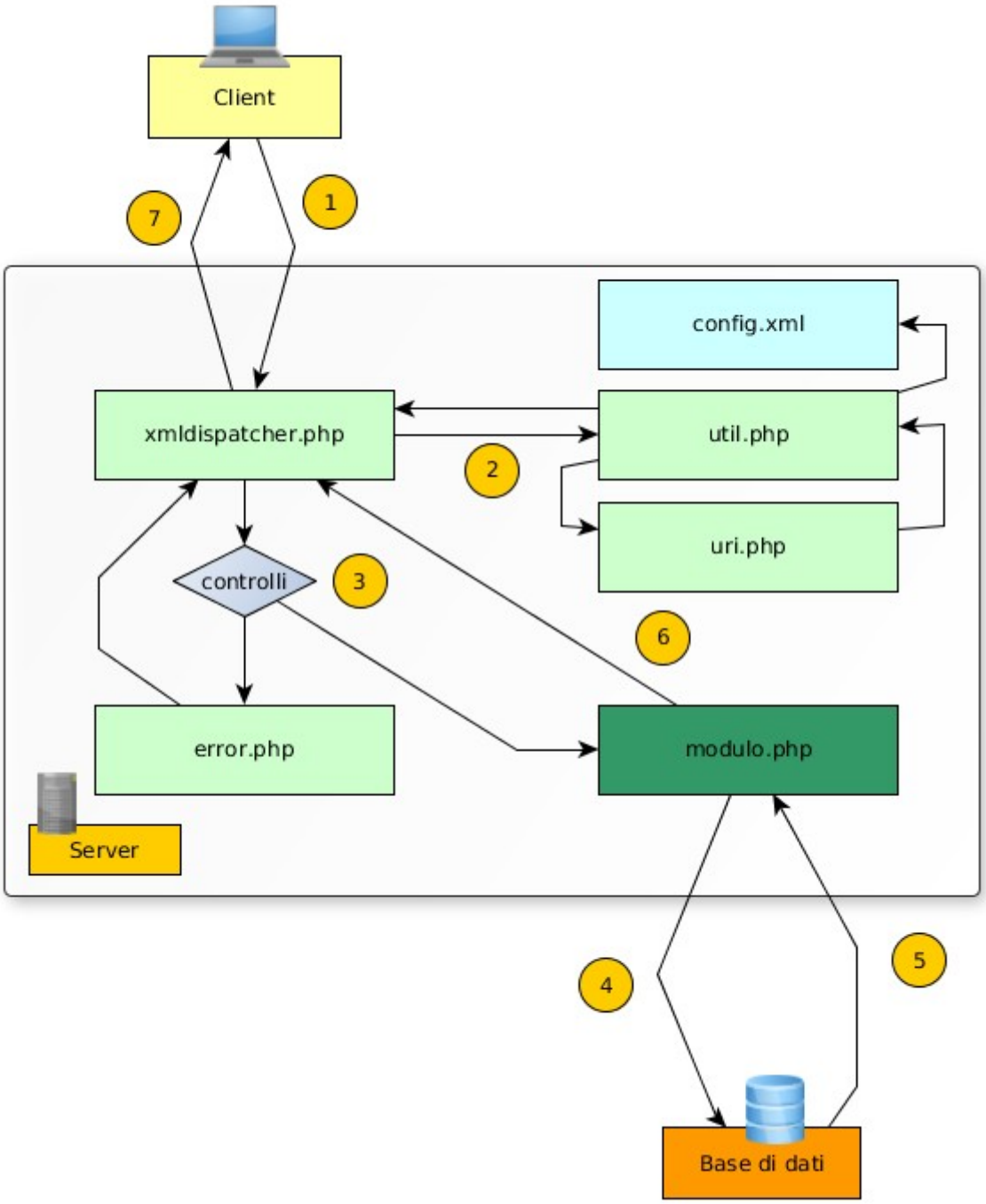


[FIG02] Sistema di gestione ricette mediche (applicazione lato client).

3.1.2 Il lato server

La parte server dell'applicazione è costituita da una serie di files scritti in *PHP* e *XML*. I files *PHP* si dividono in dispatcher, classi di utility e moduli di connessione. Ogni modulo di connessione è associato ad una specifica base di dati. I moduli di connessione verranno analizzati in dettaglio nel capitolo 4.

La figura [FIG03] mostra in dettaglio l'interoperabilità dei componenti sopra descritti.



[FIG03] Interoperabilità dei componenti lato server.

Seguendo la numerazione in figura, il processo di una richiesta *CRUD* è il seguente: una volta ricevuta una richiesta dal client *(1)*, il dispatcher istanzia gli oggetti che contengono le informazioni da passare al modulo di connessione *(2)*, dopodiché effettua una serie di controlli di validazione *(3)*. Se i controlli vengono superati, il modulo effettua una richiesta *CRUD* sulla base di dati *(4)*. La base di dati elabora la richiesta e invia una risposta codificata *(5)* al modulo, diversa a seconda del

tipo di tecnologia. Il modulo decodifica la risposta e produce un file XML che il dispatcher legge *(6)* e invia al client *(7)*.

3.2 I files *PHP*

La parte server dell'applicazione è composta dai seguenti files *PHP*:

- *index.php:* avvia l'autenticazione dell'utente (tramite *auth.php*), e inizializza il codice HTML, i fogli di stile e il codice Javascript di base.
- *auth.php:* controlla la lista degli utenti (*users.xml*) e restituisce un messaggio di errore in caso di dati non corretti, altrimenti imposta le variabili di sessione e permette il login.
- *error.php:* stampa i messaggi di errore dovuti a operazioni non consentite da parte dell'utente e/o problemi nel codice. La variabile *$errorMessage* genera il codice HTML della pagina in cui viene visualizzato l'errore.
- *uri.php:* gestisce le operazioni su *XML.*
- *util.php:* raggruppa le classi che contengono le informazioni sulle richieste da effettuare.
- *xmldispatcher.php (*detto anche semplicemente *dispatcher*)*:* Istanzia le principali classi. Fa da interfaccia tra la parte php e la parte js/xml del progetto.
- *xmlimporter.php*: gestisce l'importazione di dati da una base di dati all'altra.
- *exist.php, mysql.php, ecc...:* gestiscono le operazioni sulle varie basi di dati.

Nel prossimo capitolo verranno analizzati nel dettaglio i moduli di connessione.

Capitolo 4 – I moduli di connessione

In questo capitolo verranno analizzati i moduli di connessione (chiamati anche *pillole*), ovvero le classi *PHP* di FimoDB adibite ad effettuare operazioni di tipo *CRUD* su una base di dati, e i relativi files di configurazione *XML*. Più in dettaglio, verranno descritti:

- Struttura dei files *PHP* (funzioni e variabili).
- Struttura dei files *XML*.
- Moduli presenti al momento.
- Alcuni algoritmi interessanti.

4.1 Cenni preliminari

Come detto precedentemente, i moduli di connessione (*pillole*) sono le classi *PHP* che contengono le funzioni adibite ad effettuare operazioni *CRUD* su una determinata base di dati. Ogni modulo prende il nome della tecnologia a cui si connette (ad esempio il modulo *Mysql* gestisce le operazioni su una base di dati di tipo *Mysql* ed è contenuto nel file *mysql.php*), e contiene una classe omonima (più eventuali classi secondarie). Questa classe contiene:

- Funzioni relative alle operazioni *CRUD*.
- Funzioni di utility.
- Variabili.

La struttura (nomi delle funzioni *CRUD* e delle variabili) è identica per ogni modulo. Cambiano invece il modo in cui vengono popolate le funzioni relative alle operazioni *CRUD* e il numero di funzioni di utility.

Ad ogni modulo è associato un file di configurazione *XML* (illustrato nel paragrafo *4.3*).

Ad ogni richiesta ricevuta, il *dispatcher* (si veda il capitolo 3) istanzia un oggetto della classe

modulo, crea gli oggetti *Source* e *Request* (analizzati tra qualche paragrafo) contenenti le informazioni sulla richiesta, effettua una serie di controlli e poi invoca la funzione dell'oggetto della classe modulo relativa all'operazione *CRUD* da effettuare, inserendo come argomenti gli oggetti di cui sopra. La funzione effettua l'operazione sulla base di dati, riceve la risposta e la converte in *XML* in modo che il *dispatcher* la possa leggere.

4.2 Componenti dei moduli di connessione

Verranno ora analizzate in dettaglio le variabili, le funzioni dei moduli di connessione,

4.2.1 Variabili

Non essendo un modulo altro che una classe *PHP*, all'interno di questo possono essere inserite e cancellate variabili in modo arbitrario. Tuttavia, è necessario che almeno una variabile sia presente: la variabile *$xml*. Questa infatti contiene la risposta (in formato *xmlData*) ricevuta dall'operazione *CRUD* effettuata sulla base di dati, e viene letta dal *dispatcher* quando l'operazione è stata eseguita. Nel prossimo paragrafo verrà spiegato come viene utilizzata questa variabile all'interno delle funzioni relative alle operazioni *CRUD*.

4.2.2 Funzioni relative ad operazioni *CRUD*

All'interno di ogni modulo, ad ogni operazione *CRUD* (si veda il capitolo 2) è associata una funzione.

- *selectRecords($r,$s)*: permette di effettuare operazioni di lettura (*Read*) su una base di dati. Gli argomenti della funzione sono gli oggetti *Source* e *Request*, che verranno introdotti nel prossimo paragrafo. Indipendentemente dal tipo di tecnologia di riferimento, la procedura che il codice deve eseguire è:
  - connettersi alla base di dati.
  - decodificare la richiesta (leggendo gli oggetti *Source* e *Request*).
  - interrogare la base di dati (per mezzo di una query o una funzione apposita).
  - decodificare la risposta (convertire l'oggetto ricevuto in *XML*).

    - effettuare eventuali modifiche sull'oggetto *XML* (ordinamento, attributi, ecc..).
    - salvare l'oggetto *XML* nella variabile *$xml* .
- *insertRecords($record)*: Permette di effettuare operazioni di scrittura (*Create*) su una base di dati. La procedura che deve essere effettuata per inserire un record è:
    - connettersi alla base di dati.
    - decodificare la richiesta (da *XML* a formato compatibile per la base di dati).
    - preparare la query di inserimento.
    - eseguire la query.
    - salvare l'oggetto *XML* (relativo alla richiesta) nella variabile *$xml*.
- *removeRecords($record)*: Permette di effettuare operazioni di cancellazione (*Delete*) su una base di dati. La procedura che deve essere effettuata per cancellare un record è:
    - connettersi alla base di dati.
    - decodificare la richiesta (da *XML* a formato compatibile per la base di dati).
    - preparare la query di cancellazione.
    - eseguire la query.
    - salvare l'oggetto *XML* (relativo alla richiesta) nella variabile *$xml*.
- *replaceRecords($record)*: Permette di effettuare operazioni di aggiornamento (*Update*) su una base di dati. La procedura che deve essere effettuata per aggiornare un record è:
    - connettersi alla base di dati.
    - decodificare la richiesta (da *XML* a formato compatibile per la base di dati).
    - preparare la query di aggiornamento.
    - eseguire la query.
    - salvare l'oggetto *XML* (relativo alla richiesta) nella variabile *$xml*.

4.2.3 Funzioni di utility

All'interno del modulo possono essere inserite funzioni in modo arbitrario. Queste funzioni possono servire per frammentare la procedura relativa ad un'operazione *CRUD*, oppure per eseguire operazioni in modo ricorsivo/iterativo. Tuttavia, nessuna di queste funzioni verrà mai richiamata da parti di codice esterne al modulo (ad eccezione di quelle funzioni che accedono alla variabile *$xml*).

4.2.4 Gli oggetti Source e Request

Per effettuare operazioni con successo, le funzioni viste nel paragrafo *4.2.2* necessitano di informazioni riguardo al tipo di richiesta. Queste informazioni sono contenute in oggetti di tipo *Source* e *Request* (si veda *util.php* al capitolo 3), e in alcuni casi vengono passate alle funzioni come argomenti. In tutti i casi è sempre possibile richiamare i due oggetti in qualsiasi punto del codice scrivendo:

```
global $source;
```

per l'oggetto *Source*, mentre.

```
global $request;
```

per l'oggetto *Request*.

Verranno ora analizzati i due oggetti nel dettaglio.

4.2.4.1 Source

Un oggetto di tipo *Source* contiene le informazioni presenti nel file di configurazione relativo al modulo di connessione (si veda il paragrafo *4.3*). Da questo oggetto è possibile ricavare parte delle informazioni per interrogare una base di dati. Parametri arbitrari possono essere aggiunti nel file di configurazione e richiamati nel codice *PHP* (una volta dichiarata la variabile *$source* come mostrato nel paragrafo *4.2.4*) scrivendo (o assegnando come valore):

```
$source->nomeparametro;
```

dove *nomeparametro* va sostituito col nome del parametro interessato. La funzione *print_r($source)* mostra l'anatomia di un oggetto *Source*.

```
Source Object
(
    [raw:protected] => Array
        (
            [config] => Array
                (
                    [0] => SimpleDOM Object
                        (
                            [@attributes] => Array
                                (
                                    [name] => pazienti
                                    [id] => pazienti
                                                                    [uri]  =>
http://ext-ension.cs.unibo.it:8080/exist/rest/db/sole/people.xml
                                    [pill] => exist
```

```
                                    [contenttype] => application/xml
                                    [access] => rw
                                    [record] => persona
                                )

                        )

                )

        )

    [filename] => ../config/config.xml
    [id] => pazienti
    [name] => pazienti
    [uri] => http://ext-ension.cs.unibo.it:8080/exist/rest/db/sole/people.xml
    [pill] => exist
    [contenttype] => application/xml
    [access] => rw
    [record] => persona
    [return] =>
)
```

[COD01] anatomia di un oggetto *Source*.

Nel paragrafo *4.3* verrà mostrato l'equivalente *XML* di questo oggetto presente nel file di configurazione.

4.2.4.2 Request

La parte di informazioni restante necessaria per effettuare l'operazione è contenuta nell'oggetto *Request*. Dentro questo oggetto è possibile trovare (ad esempio) informazioni sull'ordinamento, sul limite di records da recuperare, sul corpo della richiesta in formato *XML*. Non è possibile aggiungere parametri. La funzione *print_r($request)* mostra l'anatomia di un oggetto *Request*.

```
Request Object
(
    [viewTemplate:Request:private] => {view}/@cod="{viewId}"
    [raw:protected] => Array
        (
            [filter] =>
            [query] =>
            [sort] =>
        )

    [body] =>
    [username] => doc1
    [method] => GET
    [contenttype] =>
    [pathinfo] => /curante/OF68554378/pazienti/
```

```
    [paging] => 1
    [start] => 0
    [limit] => 50
    [filter] => Array
        (
            [type] => none
            [field] =>
            [op] =>
            [value] =>
        )

    [sort] => Array
        (
        )

    [entity] => collection
    [view] => curante/@cod="OF68554378"
    [viewId] => OF68554378
    [db] => pazienti
)
```

[COD02] anatomia di un oggetto *Request*.

Questi parametri possono essere richiamati nel codice *PHP* (una volta dichiarata la variabile *$request* come mostrato nel paragrafo *4.2.4*) scrivendo (o assegnando come valore):

```
$request->nomeparametro;
```

4.3 Il file di configurazione

Come già detto in precedenza, ad ogni modulo viene associato un file di configurazione scritto in *XML*. Questo file è il ponte di connessione tra l'applicazione *client* e i moduli. Analizzando la struttura del file, è possibile notare una serie di nodi-figlio chiamati *concetti*. Questi nodi, variabili a seconda dell'applicazione *client*, contengono una serie di attributi che può essere ampliata a piacere. Questi attributi verranno poi inseriti all'interno dell'oggetto *Source* che viene istanziato ad ogni richiesta di tipo *CRUD*.
Nel paragrafo *4.2.4.1* è stata mostrata l'anatomia di un oggetto *Source*. Lo script [COD03] mostra il concetto corrispettivo presente nel file di configurazione:

```
<concept
      name="pazienti"
      id="pazienti"
      uri="http://ext-ension.cs.unibo.it:8080/exist/rest/db/sole/people.xml"
      pill="exist"
      contenttype="application/xml"
```

```
      access="rw"
      record="persona"
/>
```

[COD03] Frammento del file di configurazione relativo all'oggetto *Source* mostrato in [COD01]

Gli attributi del nodo *XML* diventano variabili dell'oggetto *Source* relativo.

4.4 Moduli esistenti

Al momento, sono disponibili moduli di connessione per le seguenti tecnologie:

- *Mysql*
- *MongoDB*
- *CouchDB*
- *PostgreSQL*
- *eXistDB*

Avendo spiegato nei paragrafi precedenti che la procedura per effettuare un operazione di tipo *CRUD* (e la relativa funzione all'interno del modulo) è sempre la stessa indipendentemente dal modulo, analizzare nel dettaglio tutte le funzioni di ogni modulo non avrebbe molto senso. Verranno quindi analizzati di seguito solo alcuni algoritmi di particolare interesse. Nel prossimo capitolo verranno invece espresse le problematiche incontrate durante lo sviluppo dei suddetti moduli.

4.4.1 Selezione dati in *Mysql*

Pur essendo progettato per funzionare su una specifica tecnologia, un modulo di connessione deve essere indipendente dalla configurazione della base di dati o da una struttura specifica. Una base di dati *Mysql* ben progettata deve rispettare i vincoli del modello *Entità/Relazione*. Recuperare dati in modo automatico (senza quindi dover scrivere query manualmente, ma utilizzando i parametri del file di configurazione ed eseguendo in modo automatico un numero arbitrario di query sulle tabelle più disparate) presenta diverse complicazioni, dovute al fatto che i dati che interessa recuperare possono risiedere anche su tabelle diverse, e che i dati, una volta convertiti in *XML*, devono essere elaborati in vari modi.

Verrà ora analizzato l'algoritmo che effettua tale operazione.

4.4.1.1 Il file di configurazione

Il frammento di codice [COD04] mostra gli attributi del concetto “prescrizioni” relativo all'applicazione client attiva al momento (un sistema di gestione di ricette mediche):

```
<concept id="prescrizioni"
      ...
      conn_host="localhost"
      conn_db="sole"
      keyfield="cod"
      attributes="cod,codice"
      codsuffix="_cod"
      origin = "paziente*pazienti/cod;dottore*dottori/cod;presso*farmacie/cod"
       originfields= "cod, cognome, nome, indirizzo, citta, citta_cod, prov, cf,
            asl, asl_cod, sesso, natoIl, natoA, cittadinanza, cittadinanza_cod,
            esenzione;cod, cognome, nome, indirizzo, citta, citta_cod, prov, cf,
            asl, asl_cod;cod, nome"
      ntonlink = "voci*voce*prescrizione_prestazione,quesito"
      ntonlinkorigin = "prestazione*prestazioni*cod/quesito*quesiti*codice"
      ntonlinkoriginfields = "codice,descrizione/codice,testo"
      ntonlinkoriginkeyfields = "codice"
      textfix= "presso/nome;quesito/testo;prestazione/descrizione"
      record="ricetta"
      fieldsOrder="paziente;dottore;data;tipoVisita;urgenza;prenotato;presso;
            eseguito;risultato;voci"
      ...
      idCount="count(//ricetta)">
```

[COD04] Concetto “prescrizioni” dentro al file di configurazione

Questa lista di attributi (per semplicità sono stati omessi gli attributi non inerenti all'algoritmo di selezione) permette di effettuare query sulla base di dati (e operazioni sulla risposta ricevuta) in modo parametrico. In particolare:

- L'attributo *origin* permette di collegare un campo (chiave esterna) della tabella principale ad un campo (chiave primaria) di una tabella secondaria. La sintassi da utilizzare all'interno degli apici è:
  *CHIAVEESTERNA1*NOMETABELLASECONDARIA1/CHIAVEPRIMARIATABSECONDARIA1;CHIAVEESTERNA2*NOMETABELLASECONDARIA2/CHIAVEPRIMARIATABSECONDARIA2;...CHIAVEESTERNAN*NOMETABELLASECONDARIAN/CHIAVEPRIMARIATABSECONDARIAN2;*

- L'attributo *originfields* permette di scegliere quali campi di ogni tabella esterna selezionare (ci deve essere congruenza con l'attributo *origin*). La sintassi da utilizzare all'interno degli apici è:
  CAMPO1TABESTERNA1,CAMPO2TABESTERNA1,CAMPONTABESTERNA1;CAMPO1TABESTERNA2,CAMPO2TABESTERNA2,CAMPONTABESTERNA2;...CAMPO1TABESTERNAN,CAMPO2TABESTERNAN,CAMPONTABESTERNAN;
- L'attributo *ntonlink* permette di collegare un campo (chiave esterna) di una tabella secondaria ad un campo della tabella principale (relazione n-a-n) La sintassi da utilizzare all'interno degli apici è:
  *NOMETABELLAESTERNA1*NOMERECORDTABELLAESTERNA1*NOMECAMPOCHIAVEESTERNATABELLAESTERNA1_CAMPO1TABELLAESTERNA1,CAMPO2TABELLAESTERNA1,CAMPONTABELLAESTERNA1;*
  *NOMETABELLAESTERNA2*NOMERECORDTABELLAESTERNA2*NOMECAMPOCHIAVEESTERNATABELLAESTERNA2_CAMPO1TABELLAESTERNA2,CAMPO2TABELLAESTERNA2,CAMPONTABELLAESTERNA2;*
  *...NOMETABELLAESTERNAN*NOMERECORDTABELLAESTERNAN*NOMECAMPOCHIAVEESTERNATABELLAESTERNAN_CAMPO1TABELLAESTERNAN,CAMPO2TABELLAESTERNAN,CAMPONTABELLAESTERNAN*
- L'attributo *ntonlinkorigin* ha la stessa funzione del campo *origin* ma riferita alle tabelle secondarie a cui sono a loro volta connesse le tabelle esterne delle relazioni n-a-n. La sintassi da utilizzare all'interno degli apici è:
  *CAMPO1TABELLAESTERNA1*NOMETABELLASECONDARIA1*CHIAVEPRIMARIATABSECONDARIA1/CAMPO2TABELLAESTERNA1*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2/CAMPONTABELLAESTERNA1*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2;*
  *CAMPO1TABELLAESTERNA2*NOMETABELLASECONDARIA1*CHIAVEPRIMARIATABSECONDARIA1/CAMPO2TABELLAESTERNA2*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2/CAMPONTABELLAESTERNA2*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2;*
  *...CAMPO1TABELLAESTERNAN*NOMETABELLASECONDARIA1*CHIAVEPRIMARIATABSECONDARIA1/CAMPO2TABELLAESTERNAN*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2/CAMPONTABELLAESTERNAN*NOMETABELLASECONDARIA2*CHIAVEPRIMARIATABSECONDARIA2;*

- L'attributo *ntonlinkoriginfields* ha la stessa funzione del campo *originfields* ma riferita alle tabelle secondarie a cui sono a loro volta connesse le tabelle esterne delle relazioni n-a-n. La sintassi da utilizzare all'interno degli apici è:
  CAMPO1TABELLASECONDARIA1ESTERNA1,CAMPO2TABELLASECONDARIA1TABESTERNA1,CAMPONTABELLASECONDARIA1TABESTERNA1/CAMPO1TABELLASECONDARIA2TABESTERNA1,CAMPO2TABELLASECONDARIA2TABESTERNA1,CAMPONTABELLASECONDARIA2TABESTERNA1/CAMPO1TABELLASECONDARIANTABESTERNA1,CAMPO2TABELLASECONDARIANTABESTERNA1,CAMPONTABELLASECONDARIANTABESTERNA1;
  CAMPO1TABELLASECONDARIA1TABESTERNA2,CAMPO2TABELLASECONDARIA1TABESTERNA2,CAMPONTABELLASECONDARIA1TABESTERNA2/CAMPO1TABELLASECONDARIA2TABESTERNA2,CAMPO2TABELLASECONDARIA2TABESTERNA2,CAMPONTABELLASECONDARIA2TABESTERNA2/CAMPO1TABELLASECONDARIANTABESTERNA2,CAMPO2TABELLASECONDARIANTABESTERNA2,CAMPONTABELLASECONDARIANTABESTERNA2;
  ...CAMPO1TABELLASECONDARIA1TABESTERNAN,CAMPO2TABELLASECONDARIA1TABESTERNAN,CAMPONTABELLASECONDARIA1TABESTERNAN/CAMPO1TABELLASECONDARIA2TABESTERNAN,CAMPO2TABELLASECONDARIA2TABESTERNAN,CAMPONTABELLASECONDARIA2TABESTERNAN/CAMPO1TABELLASECONDARIANTABESTERNAN,CAMPO2TABELLASECONDARIANTABESTERNAN,CAMPONTABELLASECONDARIANTABESTERNAN;

4.4.1.2 Il modulo PHP

Il frammento di codice [COD05] mostra l'algoritmo di selezione nel quale vengono impiegati i parametri descritti nel paragrafo precedente (per motivi di spazio verrà scritto in pseudocodice):

```
selectRecords($s,$r) {

connessione alla base di dati ($source->conn..)
selezione di tutti i record della tabella principale ($source->id)
conversione del risultato in XML
conversione di alcuni campi ($source->attributes) in attributi di altri campi

      per ogni chiave esterna ($source->origin) {

            selezione dei campi della tabella esterna ($source->originfields)
            conversione del risultato in XML
```

```
            agganciamento del nodo XML creato al nodo della tabella principale
      }

      per ogni relazione n-a-n($source->ntonlink) {

            selezione dei campi della tabella esterna

            per ogni chiave esterna della tabella esterna
($source->ntonlinkorigin) {

                  selezione dei campi della tabella esterna
($source->ntonlinkoriginfields)
                  conversione del risultato in XML
                  agganciamento del nodo XML creato al nodo della tabella
      principale
            }

            conversione del risultato in XML
            agganciamento del nodo XML creato al nodo della tabella principale
      }

riordinamento dei nodi XML ($source->fieldsorder)
conversione di alcuni campi in testo del nodo padre
salvataggio del risultato dentro alla variabile $xml

}
```

[COD05] Pseudocodice dell'algoritmo di selezione

Questo meccanismo permette di effettuare query in modo più potente rispetto a qualsiasi altra applicazione *RESTful* che supporti *Mysql* presente sul mercato. Il rovescio della medaglia è che il settaggio dei vari parametri può risultare molto complesso. Nel capitolo 5 verranno descritte le difficoltà riscontrate durante lo sviluppo (ed i punti deboli) di questa implementazione. Nel capitolo 6 verrà proposta una soluzione alternativa (meno potente ma più semplice da impostare).
L'appendice A contiene una guida completa per realizzare nuovi moduli di connessione.

Capitolo 5 – Considerazioni

In questo capitolo verranno analizzati, in merito allo sviluppo di FimoDB:

- Problemi incontrati durante lo sviluppo dei moduli di connessione.
- Punti di forza e difetti dei vari moduli di connessione.

5.1 Problematiche incontrate durante lo sviluppo

E' stato mostrato nel capitolo 2 che le operazioni di tipo *CRUD* hanno una funzione corrispettiva su ogni tecnologia. E' sempre possibile fare in modo che un'applicazione funzioni su qualsiasi base di dati. Tuttavia, questo non significa che lo sviluppo proceda sempre senza intoppi e che non ci sia una soluzione ottimale per ogni tipo di applicazione. Ogni tipo di base di dati ha una struttura e delle regole proprie. Un'operazione che risulta immediata su una tecnologia può risultare particolarmente ostica su un'altra tecnologia. Verranno ora elencate le difficoltà incontrate nello sviluppo di codice per ogni tecnologia.

5.1.1 Basi di dati classiche e tecnologia *NoSQL*

Dall'introduzione del modello e*ntità/relazione,* le basi di dati di tipo relazionale hanno sempre sofferto di una certa rigidità di struttura (per rispettare i vincoli di tale modello). Con l'introduzione della tecnologia *NoSQL* la rigidità strutturale viene superata e diventa possibile effettuare operazioni di tipo *CRUD* in modo più libero.
Ad esempio, è possibile inserire dati in un documento (l'equivalente *NoSQL* del record di una tabella) aggiungendo o togliendo campi in modo arbitrario. Su una base di dati classica, invece, gli unici campi inseribili sono quelli pre-impostati nella tabella.
Se da un lato l'utilizzo di una struttura priva di vincoli concettuali presenta più libertà, possono sorgere dei problemi di ridondanza, mantenibilità e integrità dei dati. Inoltre (come verrà mostrato nel paragrafo *5.1.3*) il sistema di query delle basi di dati *NoSQL* non è sempre potente come quello delle basi di dati classiche.

Che differenze si notano nello sviluppare un'applicazione che funziona allo stesso modo utilizzando tecnologie differenti (in questo caso agli antipodi)?

5.1.2 Selezione e inserimento dati: confronto tra *Mysql* e *MongoDB*

Nel capitolo precedente è stato mostrato un algoritmo che permette di interrogare una base di dati *Mysql* effettuando un numero di query arbitrarie in modo automatico. Il problema principale riscontrato durante lo sviluppo di tale algoritmo è che, nel caso di query complesse in cui è necessario recuperare dati presenti in tabelle diverse, la rigidità del modello *entità/relazione* complica le cose.
Come cambia la situazione effettuando le stesse richieste su una base di dati di tipo *NoSQL?*

Il frammento di codice [cod06] mostra lo pseudocodice equivalente nel modulo *MongoDB*:

```
selectRecords($s,$r) {

connessione alla base di dati ($source->conn..)
impostazione dei filtri della query ($r->filter,$r->...)
interrogazione della base di dati
ordinamento dei risultati
rimozione degli elementi superflui
conversione da JSON a XML
salvataggio del risultato dentro alla variabile $xml

}
```

[COD06] Pseudocodice dell'algoritmo di selezione

Il frammento di codice [cod07] mostra l' equivalente *MongoDB* del concetto “prescrizioni” presente nel file di configurazione visto nel capitolo precedente:

```
<concept id="prescrizioni"
      name="prescrizioni"
      ...
      contenttype="application/xml"
      record="ricetta"
      idProperty= "@cod"
      import="false"
      keyfield="cod"
      codice="true"
      listobj="voce"
      autoarray="voci"
```

```
      idValueTemplate="{Y}-{usercod}"
      idDelimiter="-"
      idCountField="ID_COUNTER"
idCount="count(//ricetta)">
```

[COD07] Concetto “prescrizioni” dentro al file di configurazione (*MongoDB*)

Da uno sguardo ai frammenti di codice risulta evidente che l'operazione di selezione sia enormemente più semplice in *MongoDB* rispetto a *Mysql*.
Questo, come detto poco fa, è dovuto al fatto che una base di dati *NoSQL* (svincolata dal modello relazionale) non contiene tabelle rigidamente strutturate, ma documenti liberamente espandibili. Siccome in un documento sono contenute tutte le informazioni necessarie a soddisfare un'interrogazione, le query risultanti sono sempre di tipo elementare.

In questo caso la tecnologia *NoSQL* dimostra di essere preferibile a quella classica, offrendo le stesse possibilità operative, ma minori difficoltà implementative. Tuttavia non è sempre così.

Nel prossimo paragrafo verrà mostrato un caso in cui la tecnologia classica risulta preferibile.

5.1.3 Selezione filtrata in *CouchDB*

Quando si effettua una richiesta di tipo *Read* su una base di dati, spesso si ha bisogno di filtrare i risultati in modo dinamico, ottenendone solo una parte. Molte tecnologie forniscono meccanismi di query sofisticati, che servono appunto a soddisfare tale esigenza. Purtroppo non è il caso di *CouchDB.*
Verranno ora analizzati il metodo con cui vengono letti i dati in *CouchDB* e la soluzione trovata per risolvere il problema della mancanza di supporto alle query dinamiche.

5.1.3.1 *CouchDb* e le *Views* (viste)

Mentre, come detto precedentemente, altre basi di dati supportano meccanismi di query dinamiche (i cui parametri vengono impostati da chi effettua la richiesta), in *CouchDB* compaiono alcuni strumenti chiamati v*iews* (viste). Una v*iew* non è altro che un documento *JSON* che contiene un frammento di codice *Javascript* (inserito nel parametro *map*) atto a mostrare i documenti presenti su una base di dati che rispettano certi criteri. Questi criteri fanno da filtro, ma devo essere

pre-impostati. Ad ogni richiesta di lettura, viene interrogata una v*iew* apposita che filtra i domumenti e restituisce un risultato.

Il frammento di codice [COD08] fornisce un esempio di v*iew.*

```
{
   "_id": "_design/farmacie",
   "_rev": "2-522456f6d222e207c423a235a9dfc16e",
   "language": "javascript",
   "views": {
       "Farmacie": {
           "map": "function(doc) {\n  emit(null, doc);\n}",
           "reduce": "_count"
       }
   }
}
```

[COD08] Esempio di *View* in *CouchDb*

Questa *view* restituisce tutti i documenti presenti nella collezione “farmacie”.
Il problema di questo sistema di interrogazione è che non accetta parametri in modo dinamico. Eventuali filtri devono essere già presenti all'interno della funzione *Javascript*. Il prossimo paragrafo mostra la soluzione implementata per superare questo limite.

5.1.3.2 Soluzione implementata

Siccome i parametri della query non possono essere inviati in modo dinamico, ma devono essere già presenti in una v*iew* contenuta nella collezione di documenti, l'unico modo possibile per aggirare questo ostacolo è il seguente: ad ogni richiesta di lettura con filtri, inserire nella collezione (prima della lettura) una v*iew* contenente i filtri, interrogare la v*iew* appena inserita, ricevere la risposta e cancellare la v*iew.* [Nic13]

Il frammento di codice [COD09] mostra lo pseudocodice della funzione *selectRecords* in C*ouchDB:*

```
selectRecords($s,$r) {

connessione alla base di dati ($source->conn..)
impostazione dei filtri della query ($r->filter,$r->...)
inserimento della vista nella collezione
interrogazione della vista appena inserita
ordinamento dei risultati
rimozione degli elementi superflui
```

```
rimozione della vista appena inserita
conversione da JSON a XML
salvataggio del risultato dentro alla variabile $xml

}
```
[COD09] Pseudocodice della funzione selectRecords in CouchDB

In questo modo è possibile filtrare i documenti. Il problema di questa implementazione è la lentezza: ad ogni richiesta vengono effettuate tre operazioni (lettura, inserimento,cancellazione). Purtroppo al momento non sembra essere disponibile una soluzione migliore.

5.2 Sintesi dei punti deboli di ogni modulo sviluppato

I moduli *Mysql*/*PostgreSQL* presentano i seguenti difetti:

- Difficoltà nel settaggio dei parametri.
- Alcune query di selezione particolarmente complesse potrebbero non funzionare.

Il modulo M*ongoDB* non ha difetti rilevanti.
Il modulo *CouchDB* avrebbe bisogno di un metodo di selezione più efficiente.

Nel prossimo capitolo verrà fornita una serie di linee guida per un'implementazione alternativa di questi moduli.

Capitolo 6 – Conclusioni

Questo capitolo conclude la trattazione e delinea il punto della situazione in merito allo sviluppo di FimoDB. Verranno discussi:

- Spunti per implementazioni alternative
- Sviluppi futuri e possibilità di ampliamento

6.1 Implementazioni alternative

Nei capitoli precedenti sono stati analizzati i moduli di connessione di FimoDB. E' stato mostrato che il modulo *Mysql* e il modulo *CouchDB* hanno dei punti deboli che potrebbero essere migliorati seguendo approcci diversi. Nei prossimi paragrafi verranno introdotte linee guida per implementazioni alternative che potrebbero risultare migliori di quelle attuali. Va precisato che queste implementazioni potrebbero anche far sorgere nuove problematiche.

6.1.1 Selezione dati in Mysql

L'implementazione attuale del modulo *Mysql* consente di effettuare interrogazioni multiple su una base di dati senza dover scrivere query in linguaggio *SQL*, ma inserendo una serie di parametri nel file di configurazione. Lo svantaggio di questo approccio è che, nel caso di interrogazioni complesse, diventa complicato settare correttamente il file di configurazione.

Una soluzione alternativa potrebbe essere quella di inserire direttamente la query *SQL* dentro al file di configurazione.

Il frammento di codice [COD10] mostra gli attributi del concetto “prescrizioni” (lo stesso descritto nel capitolo 4) all'interno del file di configurazione nel caso di un:

```
<concept id="prescrizioni"
      ...
      conn_host="localhost"
```

```
      conn_db="sole"
      keyfield="cod"
      attributes="cod,codice"
      codsuffix="_cod"
      query_read="..."
      query_create="..."
      query_delete="..."
      query_update="..."
      record="ricetta"
      ...
      idCount="count(//ricetta)">
```

[COD10] Concetto “prescrizioni” dentro al file di configurazione

E' possibile notare che i parametri dell'implementazione precedente sono stati sostituiti con quattro parametri *query_CRUD* in cui andrebbe eventualmente inserita la query relativa ad ogni operazione. Se è necessario ottenere informazioni da più tabelle medianti chiavi esterne, bisognerà utilizzare la clausola *JOIN* collegando le tabelle tra di loro.

Questa implementazione, sebbene semplifichi il file di configurazione, fa sorgere le seguenti problematiche:

- Bisogna trovare un modo per effettuare query dinamiche. Una possibile soluzione potrebbe essere inserire all'interno dei parametri *query_CRUD* un misto tra linguaggio *SQL* e variabili *PHP* che assumeranno un valore diverso a seconda della richiesta effettuata.

- Il codice *XML* prodotto a partire dalla risposta deve essere modificato prima che venga letto dal *dispatcher*: alcuni nodi devono diventare attributi di altri nodi, altri nodi devono essere rimossi, altri ancora devono diventare testo, ecc.. . Nel caso di query complesse con clausole (JOIN multiple), potrebbe diventare complicato effettuare queste operazioni.

Il frammento di codice [COD11] mostra lo pseudocodice di questa implementazione della funzione *selectRecords* in Mysql*:*

```
selectRecords($s,$r) {

connessione alla base di dati ($source->conn..)
esecuzione della query ($source->query_read)
conversione del risultato in XML
conversione di alcuni campi ($source->attributes) in attributi di altri campi
riordinamento dei nodi XML ($source->fieldsorder)
conversione di alcuni campi in testo del nodo padre
```

```
salvataggio del risultato dentro alla variabile $xml

}
```

[COD11] Pseudocodice della funzione selectRecords in *Mysql* (implementazione alternativa)

E' evidente che rispetto all'implementazione attuale il codice risulta più semplice e pulito.

6.1.2 Query dinamiche in *CouchDB*

E' stato mostrato nel capitolo 5 che *CouchDB* non fornisce un meccanismo per effettuare query dinamiche. La soluzione trovata per aggirare il problema consiste nell'inserire una *view* contenente i parametri richiesti, interrogare la *view* appena inserita e infine cancellarla. Il problema di questa soluzione è la velocità di esecuzione: vengono effettuate tre operazioni *CRUD* per ogni lettura.

Un'altra soluzione potrebbe essere utilizzare un motore di ricerca interno come *Apache Lucene*, che permette di ricercare parti di testo all'interno di una basi di dati. [Luc14]

Utilizzato in combinazione con *CouchDB*, all'interno di un documento *JSON* come quello mostrato nel frammento di codice [COD12], dovrebbe essere possibile filtrare i risultati in modo dinamico facendo una ricerca del tipo *"parametro": "valore"*.

```
{
   "_id": "e4a900585da828c7ad5b1330f4080886",
   "_rev": "1-487315764898f0c288ba6470a2a19e67",
   "cod": "far0124",
   "nome": "Zincone",
   "indirizzo": "Via Sardegna 1, Bologna BO, Italy",
   "telefono": "051 491008",
   "note": [
   ],
   "lat": "44.476527",
   "long": "11.383253",
   "aperto": "Mon, Tue, Wed, Thu, Fri: 0830-1230 1530-1930.",
   "ID_COUNTER": "126"
}
```

[COD12] Esempio di documento JSON in CouchDB

6.2 Sviluppi futuri e possibilità di ampliamento

FimoDB è funzionante ed utilizzabile per una grande varietà di progetti. Tuttavia, l'applicazione potrebbe essere migliorata in vari modi. Di seguito verranno presentate alcune idee.

6.2.1 Popolazione di funzioni

Tutti i moduli attualmente sviluppati hanno all'interno alcune funzioni vuote, che andrebbero popolate. Queste funzioni sono:

- *modifyRecords()*: è simile a *replaceRecords()* ma riguarda il metodo HTTP *PATCH*.
- *archiveRecords():* è simile a *removeRecords()* ma invece di cancellare i records, li archivia.

Queste due funzioni vengono richiamate dal *dispatcher* allo stesso modo di quelle già presenti. A seconda della tecnologia per cui si sta sviluppando, bisogna trovare una soluzione implementativa diversa.

6.2.2 Aggiunta di moduli

E' possibile aumentare a piacere il supporto di tecnologie, implementando nuovi moduli. Le tecnologie consigliate (in quanto più utilizzate) sono:

- *Oracle*
- *MS SQL*
- *Cassandra*
- *Access*
- *SQLite*

6.3 Sintesi finale

E' stata analizzata la struttura di FimoDB, ovvero una parte client che può assumere svariate

sfaccettature, e una parte server che può connettersi a varie basi di dati. E' stato mostrato che, rispetto alle soluzioni presenti sul mercato, fornisce il supporto a tecnologie che trascendono il modello relazionale, che è facilmente espandibile attraverso l'implementazione di nuovi moduli di connessione, che implementa algoritmi di interrogazione potenti. Sono state proposte implementazioni alternative, che possono essere migliori per certi versi ma anche portare nuove complicazioni. Infine sono state mostrare le possibilità di ampliamento (aggiunta di nuovi moduli e nuove funzioni).

FimoDB è un'applicazione web *RESTful database-independent* competitiva e con un grande potenziale.

Bibliografia

[Fie00] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures,* 2000 *http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm*

[Joo13] Joomla.org, *Technical Requirements*, ultima visita 04/02/2014, http://www.joomla.org/technical-requirements.html

[HH13] Drupal.org, Gábor Hojtsy, Lee Hunter, *Community Documentation,* 18/12/2013, *https://drupal.org/requirements*

[Wor13a] Wordpress.org, *Requirements*, ultima visita 04/02/2014, http://wordpress.org/about/requirements/

[Osc13] Oscommerce.com, *Tecnical Requirements*, ultima visita 04/02/2014, http://www.oscommerce.com/Products

[Mag13] Magento.com, *Tecnical Requirements*, ultima visita 04/02/2014, http://magento.com/resources/system-requirements

[Wor13b] Wordpress.org, *Using Alternative Databases*, ultima visita 04/02/2014, http://codex.wordpress.org/Using_Alternative_Databases

[SV99] Carl Shapiro, Hal Varian, *Information Rules, Etas Libri, 1999*

[Wor13c] Wordpress.org, *Wordpress sites in the world*, ultima visita 04/02/2014 http://en.wordpress.com/stats/

[AT12] Oracle.com, *Database Independence Vs Database Dependence*, 14/04/2012 http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:4694004400346543034

[TF12] Todd Friedrich, *What is REST?*, 2012 http://www.restapitutorial.com/lessons/whatisrest.html

[WP14] Wikipedia, *Tavola CRUD*, ultima visita 04/02/2014, http://it.wikipedia.org/wiki/Tavola_CRUD

[Bas14] Basex.org, *Xquery Update*, ultima visita 04/02/2014, http://docs.basex.org/wiki/XQuery_Update

[Cho12] Arnab Roy Chowdhury, *Database Indipendent Application*, 19/04/2012, http://code.msdn.microsoft.com/windowsdesktop/Database-Indipendent-8ab7bbba

[Sla14a] Slashdb.com, *How it works*, ultima visita 04/02/2014, http://www.slashdb.com/how-it-works/

[Sla14b] Slashdb.com, *Top 10 Database league table*, ultima visita 04/02/2014, http://www.slashdb.com/tag/relational/

[Res14] Restsql.org, *Overview*, ultima visita 04/02/2014, http://restsql.org/doc/Overview.html

[Dbe14] Db-engines.com, *DB-Engines Ranking*, ultima visita 04/02/2014, http://db-engines.com/en/ranking

[Nic13] Daniele Niccià, *L'integrazione di basi di dati NoSQL in un'applicazione web: esperienze e confronti, Università di Bologna, 2013*

[Luc14] lucene.apache.org, *Lucene Features*, ultima visita 19/02/2014, http://lucene.apache.org/core/

Appendice A - Guida alla realizzazione di nuove pillole

Prerequisiti

Per creare una nuova pillola con successo, è necessario soddisfare i seguenti requisiti:

- Avere buone basi di PHP (e in generale avere chiari i concetti del paradigma OOP).
- Avere buone basi di XML.
- Avere chiaro (a grandi linee) il funzionamento delle altre parti di codice del progetto. Non verrà spiegato in questa sede come funzionano le altre parti di codice PHP, o come funziona la parte client, tranne quando non sarà strettamente necessario.
- Avere chiari i principi del modello REST.

Definizioni

Pillola: modulo di connessione tra il database e la parte client del progetto. Permette di effettuare operazioni di tipo CRUD sul database. E' completamente indipendente dal tipo di contesto e dall'applicazione. E' scritta in PHP. E' costituita da una classe principale, più eventuali classi secondarie. La classe principale contiene variabili e funzioni relative alle operazioni CRUD, più eventuali funzioni di utility. Le classi secondarie contengono variabili e funzioni che vengono richiamate dalla classe principale. Le classi possono essere in un file unico, oppure in file diversi (tuttavia è necessario che sia presente un file principale che includa gli altri).

Files di configurazione: documenti XML, variabili a seconda del tipo di contesto e di applicazione realizzata. Contengono i parametri che vengono passati alla pillola e che servono per effettuare le operazioni di tipo CRUD.

Oggetto Source: oggetto PHP che contiene i parametri del file di configurazione. Funziona da intermediario tra la pillola e il file di configurazione.

Oggetto Request: oggetto PHP che contiene la richiesta del client in formato XML e i filtri della richiesta.

CRUD: acronimo di Create-Read-Update-Delete (crea-leggi-aggiorna-cancella). Le quattro operazioni di base che possono essere effettuate su un database.

Database (o base di dati): archivio di informazioni organizzate secondo un modello logico.

PHP: linguaggio di programmazione ad oggetti Server-Side.

XML: linguaggio di markup che consente di archiviare informazioni in una struttura ad albero.

REST: acronimo di Rapresentational State Transfer, modello architetturale pensato per applicazioni web. Per una spiegazione approfondita leggere
http://it.wikipedia.org/wiki/Representational_State_Transfer .

Ubicazione delle pillole

Le pillole sono (e vanno) situate all'interno della directory:

[RADICE DEL PROGETTO]/doc/php

Ad esempio, la pillola "dbgenerico" avrà come indirizzo del file principale:

[RADICE DEL PROGETTO]/doc/php/dbgenerico.php

Sebbene non sia obbligatorio, per motivi pratici è consigliabile dare al file PHP principale il nome della pillola.

Ubicazione del file di configurazione

I file di configurazione si trovano all'interno della directory:

[RADICE DEL PROGETTO]/doc/config

Non possono esserci più file di configurazione attivi contemporaneamente. Il file di configurazione attivo ha come indirizzo:

[RADICE DEL PROGETTO]/doc/config/config.xml

Sebbene non sia obbligatorio, per motivi pratici è consigliabile tenere tutti i file di configurazione nella stessa cartella.

Flussi di lavoro

Una pillola è un anello della catena che permette all'utente di eseguire un'operazione CRUD. Il flusso di lavoro dentro cui è situata la pillola è il seguente:

UTENTE -> RICHIESTA (CRUD) -> LATO CLIENT (Javascript) -> LATO SERVER (xmldispatcher.php) -> PILLOLA -> DATABASE -> PILLOLA (output risultato in XML) -> LATO SERVER (xmldispatcher.php) -> LATO CLIENT -> UTENTE

Quindi, una pillola riceve in input una richiesta (in formato XML o come oggetto di tipo Request), dei parametri (oggetto di tipo Source), accede al database, decodifica la risposta del database e fornisce il risultato in formato XML. Tutto questo avviene dentro una funzione specifica. Ad esempio, una richiesta di lettura (Read) seguirà questo flusso di lavoro:

RICHIESTA -> FUNZIONE (selectRecords($request,$source)) -> CONNESSIONE AL DATABASE -> LETTURA PARAMETRI PER EFFETTUARE LA RICHIESTA (oggetti Source e Request) -> INVIO DELLA RICHIESTA AL DATABASE -> DECODIFICA DELLA RISPOSTA (conversione in XML) -> OUTPUT ($this->xml = new xmlData([STRINGA CHE CONTIENE L'XML]);)

Una richiesta di inserimento (Create) seguirà invece questo flusso:

RICHIESTA -> FUNZIONE (insertRecords($request)) -> CONNESSIONE AL DATABASE -> LETTURA PARAMETRI PER EFFETTUARE LA RICHIESTA (oggetti Source e Request) -> INVIO DELLA RICHIESTA AL DATABASE -> OUTPUT ($this->xml = new xmlData([STRINGA CHE CONTIENE L'XML]);)

Realizzare una pillola partendo da un template

Il template

Il file da utilizzare come template per creare una nuova pillola si chiama "template.php" ed è presente nella directory

[RADICE DEL PROGETTO]/doc/php.

Il template è composto da una classe chiamata provvisoriamente “template” (che andrà rinominata con il nome della pillola che si vuole creare) che contiene:

- funzioni principali, relative alle operazioni CRUD
- funzioni di utility
- alcune variabili

Le funzioni associate alle operazioni CRUD verranno descritte nel paragrafo successivo. Le

funzioni di utility vengono richiamate dalle funzioni principali per svolgere determinate operazioni (come la conversione di oggetti, la ricerca di elementi, ecc..) e possono essere create in modo arbitrario. Non è detto che le funzioni già esistenti nel template siano indispensabili. Lo stesso discorso vale per le variabili, tuttavia le tre variabili *“$uri”*, *“$params”*, *“$xml”*, dichiarate all'inizio della classe, sono importanti e non vanno cancellate. In particolare, la variabile $xml è quella in cui andrà salvato l'output XML alla fine di ogni funzione.

Le operazioni possibili (e le relative funzioni)

Questa è la lista delle operazioni che una pillola può effettuare. Ogni operazione ha una funzione relativa, che deve essere presente nella pillola:

Operazione: *GET*
Funzione: *selectRecords($request,$source)*

Operazione: *PUT*
Funzione: *replaceRecords($request->finalBody)*

Operazione: *POST*
Funzione: *insertRecords($request->finalBody)*

Operazione: *PATCH*
Funzione: *modifyRecords($request->finalBody)*

Operazione: *DELETE*
Funzione: *removeRecords($request->finalBody)*
(oppure) *archiveRecords($request->finalBody)*

Queste operazioni vengono gestite dal file xmldispatcher.php, che istanzia di volta in volta un oggetto sotto forma di pillola e richiama le sue funzioni. Perchè le operazioni funzionino, è necessario popolare queste funzioni.

**consiglio: a volte è necessario richiamare una funzione più volte durante la stessa richiesta. Dato che una soluzione ricorsiva non è sempre possibile o conveniente, una buona idea è creare una funzione subordinata (che esegue effettivamente l'operazione) da richiamare quante volte si vuole dentro la funzione principale.*

La connessione al database

Per effettuare le operazioni, è necessario disporre delle credenziali di accesso al relativo database. Generalmente, i dati di cui bisogna disporre sono: username, password, host, nome del database. Le funzioni PHP che permettono di stabilire la connessione cambiano a seconda del tipo di database. A volte i moduli PHP che permettono di interfacciarsi con il database sono pre-installati (come nel caso di MYSQL). Altre volte è necessario installarli manualmente. E' necessario documentarsi leggendo le guide specifiche.
La connessione può essere effettuata all'inizio di ogni funzione, oppure quando viene istanziata la pillola (dentro ad una delle cosidette "funzioni magiche" di PHP).

**consiglio: la soluzione ottimale consiste nel creare una funzione apposita in cui effettuare la connessione, e richiamare quella funzione all'inizio di ogni funzione relativa ad un'operazione. Inoltre, conviene dichiarare i parametri di connessione (magari in un array) all'inizio della classe. In questo modo si evita di riscrivere la stessa parte di codice ogni volta, e sarà più facile modificare il codice in futuro.*

Ottenere i parametri: gli oggetti Source e Request

Una volta effettuata la connessione, è necessario ottenere i parametri specifici per soddisfare la richiesta dell'utente. Gli oggetti Source e Request servono a questo proposito.

Source

Un oggetto di tipo Source contiene tutte le informazioni presenti nel file di configurazione (config.xml) relativo alla pillola. Dentro a quel file vanno scritte quindi (ad esempio) le informazioni relative alle tabelle del database, ai campi con proprietà particolari, ecc.
L'oggetto viene istanziato ad ogni richiesta dentro xmldispatcher.php. E' possibile accedervi in qualsiasi punto del codice scrivendo:

global $source;

Se dentro a config.xml è presente un nodo come il seguente:

```
<concept id="concetto"
name="nomeconcetto"
uri="http://indirizzo"
pill="dbgenerico"
keyfield="codice"
tabella="nometabella"
...
parametro="valore"
>
```

sarà possibile accedere a queste informazioni dentro all'oggetto $source, scrivendo nel codice PHP:

```
$source->id;
$source->name;
$source->uri;
$source->pill;
$source->keyfield;
$source->tabella;
$source->parametro;
```

Nuovi parametri possono essere aggiunti e cancellati in modo arbitrario dentro al file di configurazione. Alcuni parametri come *"uri","id","name"* sono necessari per altre parti del codice e non vanno cancellati. Se non servono, è sufficiente lasciarli vuoti.

**consiglio: per rendere il codice PHP più snello, semplice e riutilizzabile possibile, conviene inserire più informazioni possibili dentro al file di configurazione, e richiamarle con il metodo*

appena descritto.

Request

Un oggetto di tipo Request contiene informazioni riguardo al tipo di operazione che deve essere effettuata, a chi effettua la richiesta e al tipo di filtro da utilizzare. Parametri interessanti sono ad esempio *"pathinfo","filter"* o *"viewId"*. Non esiste un file di configurazione che permetta di modificare dall'esterno i parametri. E' possibile accedere all'oggetto in qualsiasi punto del codice scrivendo:

global $request;

E successivamente:

$request->parametro;

Per sapere cosa contiene l'oggetto in modo specifico, basta inserire nel codice:

print_r($request);

ed effettuare una richiesta nel browser. Verranno stampati tutti i parametri in modo leggibile.

Interrogazione del database

Ottenuti i parametri necessari, si può procedere ad effettuare l'operazione sul database. Le funzioni relative alle operazioni cambiano a seconda del database. E' necessario leggere le guide specifiche.

Processare i dati:

Se l'operazione da eseguire è di tipo Read, quindi GET, una volta interrogato il database è necessario trattare i dati per renderli leggibili dal client. La risposta del database va decodificata tramite opportune funzioni PHP, diverse a seconda del tipo di database. E' necessario leggere le guide specifiche.

**consiglio: gli oggetti più pratici da utilizzare per la conversione sono quelli di tipo SimpleXmlElement. Dispongono di tutte le funzioni necessarie per le operazioni su XML.*

Output in XML

Il passaggio finale consiste nel preparare una stringa XML e inserirla in un oggetto xmlData. Tutte e sei le funzioni principali devono avere come ultima linea di codice (prima della parentesi di chiusura):

$this->xml = new xmlData(STRINGA CONTENENTE I DATI IN FORMATO XML);

Tra le parentesi va inserito il codice XML in formato stringa. Se è stato utilizzato un oggetto di tipo SimpleXMlElement, contenuto in una variabile chiamata (ad esempio) *$sxe*, sarà possibile

utilizzare la funzione *asXML()* che restituisce il contenuto dell'oggetto in formato stringa. La linea di codice diventerà:

$this->xml = new xmlData($sxe->asXML());

Errori da evitare

Violare le condizioni del paradigma REST

Le pillole fanno parte del lato server del progetto. Il lato server deve rimanere totalmente indipendente dal lato client, e deve funzionare a prescindere dal tipo di applicazione sviluppata. Pertanto, nelle pillole non deve comparire alcun riferimento all'applicazione. Se ad esempio l'applicazione è un sistema di gestione di una biblioteca, concetti come “libro”,”autore”, “data di ritorno” non devono comparire direttamente all'interno del codice PHP, ma vanno inseriti nel file di configurazione, che ha la funzione di intermediario.

Modificare altre parti di codice (sia server che client) per far funzionare la pillola

La pillola deve essere concepita in modo da prevedere il più ampio numero di configurazioni possibili. Se ad ogni modifica del resto del progetto (che sia l'applicazione client o un'altra parte del codice PHP) la pillola non regge i cambiamenti, vuol dire che è stata progettata male e sarebbe una buona idea riscrivere il codice.

Forzare il contenuto del database per adattarlo alla pillola

Ogni pillola è progettata per funzionare con un tipo specifico di database. Ogni tipo di database ha le sue regole di progettazione e struttura dei dati (ad esempio, i database relazionali ben progettati devono seguire le linee guida del modello E/R). A volte può capitare che queste regole rendano complicata la comunicazione tra il lato client e il database, e apparentemente potrebbe sembrare una buona idea infrangerle per semplificare la progettazione della pillola. A lungo termine però questa scelta non ripagherà: gli aggiornamenti e le modifiche risulteranno difficoltosi, se non impossibili. Conviene investire più tempo nella progettazione della pillola e rispettare le regole.

Riassunto

Le pillole sono anelli di congiunzione tra la parte client del progetto e i vari database. Contengono le funzioni che permettono di effettuare operazioni di tipo CRUD. Ad ogni richiesta, viene istanziata una nuova pillola (sotto forma di oggetto), dopodiché viene richiamata una sua funzione (relativa all'operazione che si vuole eseguire).
Perché una pillola funzioni è necessario popolare le funzioni relative ad ogni operazione. Per fare questo, dentro ad ogni funzione deve essere effettuate la connessione al database, l'interrogazione del database e la conversione del risultato dell'interrogazione in XML. In questo modo il client otterrà la risposta alla sua richiesta.