UNIVERSITÀ DEGLI STUDI DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

Estensione e validazione di CHKB, una base di conoscenza web per il Museo degli Strumenti per il Calcolo

Laureando:
Enrico Meloni

I Relatore:
Ing. Giuseppe Lettieri

II Relatore:
Giovanni Cignoni

17 luglio 2016

Indice

Capitolo 1

Introduzione

La tesi ha previsto come obbiettivo primario la correzione di diversi problemi presenti nel sito web CHKB[1] e la sua successiva estensione con funzionalità necessarie ad una migliore esperienza dell'utente. CHKB è una base di conoscenza pensata per raccogliere tutte le informazioni disponibili sugli strumenti per il calcolo da collezioni pubbliche e private. Queste informazioni sono poi presentate al pubblico con diversi livelli di dettaglio in modo da rendere piacevole la navigazione sia all'appassionato sia a chi ha poca conoscenza del settore. Questa relazione sarà strutturata come segue: il primo capitolo parlerà dello stato in cui si trovava il software prima del mio intervento e descriverà brevemente le tecnologie sfruttate nella realizzazione del sito; il secondo capitolo esaminerà i problemi che si sono riscontrati durante la validazione; il terzo capitolo esporrà le soluzioni studiate per risolvere le mancanze messe in luce nel capitolo precedente; il quarto capitolo esaminerà i risultati ottenuti; il quinto parlerà di possibili sviluppi e migliorie pensate durante i test d'utilizzo del sito e durante lo sviluppo.

[1] Computer History Knowledge Base

Capitolo 2

Base di partenza

La base di partenza era composta dal sito stesso, sviluppato con PHP, JavaScript, HTML e CSS. Il codice è organizzato secondo il Design Pattern MVC. Il livello di presentazione è gestito tramite il framework Bootstrap. Il progetto e lo sviluppo del sito si è svolto in più fasi da parte di persone diverse nell'ambito di più tesi o tirocini. Nel seguito si darà una breve descrizione del codice già esistente, mentre per un'analisi più approfondita del progetto del nucleo centrale del software si rimanda alla tesi di A. Biagini[1], mentre per il progetto della funzionalità dei percorsi si rimanda alla relazione del tirocinio di D. Aimini[2].

Gli utenti possono essere divisi in tre categorie principali:

Revisori : sono gli utenti che possono assegnare in redazione una scheda informativa ad un utente e che possono accettare o rifiutare le modifiche fatte;

Redattori : sono gli utenti a cui viene assegnata una scheda informativa e che sono in grado di modificare le informazioni in essa presenti. Ad un qualunque utente può essere dato l'incarico di redattore;

Utenti semplici : sono gli utenti che non possono né redarre né revisionare una scheda informativa.

Gli strumenti di calcolo vengono rappresentati in CHKB attraverso una Scheda Informativa. Ogni Scheda Informativa ha associate Versioni diverse, che rappresentano stadi di modifica da parte dei Redattori e di convalida da parte dei Revisori. Un Revisore può assegnare in redazione una Versione ad un Utente, rendendolo Redattore per quella versione della Scheda Informativa, mentre le altre restano inalterate. In questo modo, se sorgessero problemi si

può tornare indietro a una Versione precedente considerata corretta. Le modifiche fatte ad una Scheda Informativa vanno poi approvate da un Revisore che ne valuterà la correttezza.

I Reperti sono gli strumenti di calcolo fisici posseduti da un collezionista o da un museo, e possono essere descritti per la loro funzionalità o stato di conservazione. CHKB offre anche la possibilità di catalogare secondo la Convenzione adottata dalla Collezione. Le collezioni e i musei possono essere registrate nella base di conoscenza, specificando la locazione, una descrizione del contenuto e le Convenzioni per la Catalogazione e l'Esposizione. Ad ogni Collezione è assegnato uno e un solo Responsabile e zero o più Catalogatori. Il Responsabile è colui che si occupa di modificare le informazioni della Collezione, mentre il Catalogatore è chi si occupa di aggiungere Reperti e inserirne i dati di Catalogazione e di Esposizione.

Le Convenzioni sono un insieme di Attributi che possono essere di vario tipo: Testo, Numerico, Intervallo Numerico, Enumerazione. Una volta stabilita una Convenzione per la Catalogazione e per l'Esposizione possono essere aggiunti Reperti alla Collezione; da quel momento in poi la Convenzione non può più essere modificata, a meno che non vengano prima cancellati tutti i Reperti; questo perché se si cambiasse avendo ancora un Reperto esistente le sue informazioni di Catalogazione o Esposizione potrebbero non essere coerenti con quelle nuove.

I Contenuti Digitali sono immagini, software, audio, documenti o altri tipi di file che possono essere associati ad un Reperto, nel qual caso sono detti Specifici, o ad una Scheda Informativa, nel qual caso sono detti Generici.

Capitolo 3

Problemi

3.1 Presentazione

Il problema che primo fra tutti saltava all'occhio era l'incoerenza fra la presentazione delle varie pagine. Alcune strutturavano i dati utilizzando gli stili forniti da Bootstrap, altre invece utilizzavano stili di default. Alcuni dati erano presentati in maniera illeggibile e disorganizzata. Altre pagine erano incomplete e presentavano diversi errori di esecuzione. Questi problemi rendevano la navigazione molto confusa e l'utente era spesso impossibilitato a effettuare le operazioni volute. Un secondo problema, rilevato dopo la lettura del documento di progetto, era che diverse specifiche sulle operazioni effettuabili dall'utente non erano state rispettate. Alcune, come la funzione di proposta di creazione di una scheda informativa, erano corrette ma non erano accessibili dall'utente, altre, come la visualizzazione dei reperti di una collezione, avevano un comportamento incorretto.

3.2 Codice

Altri problemi erano presenti nel codice che nonostante eseguisse correttamente non seguiva il modello MVC rendendolo poco manutenibile. Un esempio era la procedura di assegnamento di responsabili e catalogatori: a seconda che questo assegnamento venisse fatto dalla pagina di modifica di un utente o dalla pagina di modifiche di una collezione esso veniva attuato da due sezioni di codice diverse, nonostante si tratti della stessa operazione. In particolare, se eseguita dalla pagina di modifica di un Utente, la sezione di codice interessata modificava direttamente la tabella delle Collezioni, mentre questo compito dovrebbe essere delegato alla classe che rappresenta le Collezioni. A volte la nomenclatura, soprattutto nella parte relativa ai percorsi, non era

chiara e utilizzava nomi non esplicativi, come ad esempio la pagina che si occupa di mostrare i percorsi: erano presenti quattro funzioni che effettuavano operazioni molto diverse con lo stesso nome, distinte solo da un numero alla fine del nome (listInfos1, listInfos2, etc.), e altre funzioni il cui nome non spiegava lo scopo delle stesse.

3.3 Estensione

Durante l'analisi e la validazione del software si è resa evidente la mancanza di alcune funzionalità necessarie:

- La prima, relativa alla navigazione, è legata al menu laterale; esso non tiene traccia della pagina in cui si trova l'utente mostrando la stessa struttura statica indipendentemente dalla posizione dell'utente. Si è ritenuto necessario rendere il menu dinamico, rendendo attivo il link relativo alla pagina in cui ci si trova;

- La seconda funzionalità si è mostrata necessaria durante le prove della funzionalità del processo di redazione e revisione. L'utente, qualsiasi fosse il suo ruolo, non aveva nessun modo di accedere alla cronologia delle sue azioni, malgrado queste fossero salvate nel database. Quindi si è deciso di aggiungere la Cronologia delle Attività, in cui l'utente può osservare le proprie azioni sulle Schede Informative.

- La terza funzionalità si è anch'essa mostrata necessaria durante le prove del processo di redazione e revisione. Il Revisore ha il potere di accettare o rifiutare una versione redatta. Tuttavia non era prevista la possibilità di spiegare le motivazioni della propria decisione. Questo è stata ritenuta una grave mancanza del processo di revisione, in quanto la decisione sarebbe dovuta essere giustificata tramite mezzi esterni al portale o lasciata immotivata. Si è deciso quindi di introdurre la possibilità di specificare un commento alla propria decisione; questo è stato esteso anche al processo di proposta di una Scheda Informativa per le stesse ragioni.

- Anche la quarta funzionalità si è mostrata necessaria nello stesso frangente. In particolare, questa necessità si ha da parte del Redattore. Precedentemente, quando una redazione veniva accettata o rifiutata, il redattore non aveva nessun modo di sapere l'esito della revisione se non controllare personalmente le modifiche aprendo la pagina della Scheda Informativa interessata. Questo metodo oltre a essere prono a errori è

anche poco pratico. Si è dunque pensato di introdurre un sistema di notifiche che avverte l'utente dell'esito della revisione. Si è poi deciso di estendere questo sistema anche alle proposte di creazione di una Scheda Informativa, all'assegnamento o alla rimozione del ruolo di Catalogatore o di Responsabile di una collezione, e all'aggiunta o rimozione di capability all'utente.

3.4 Capabilities

Un problema meno evidente era quello legato alle capabilities, ossia le azioni che un dato utente è autorizzato ad effettuare, che spesso manifestavano comportamenti scorretti. Tra pagine diverse si avevano dei comportamenti incoerenti, dovuti al modo in cui sono stati implementati i controlli sulle capabilities. Il modello Utente ha delle proprietà di tipo boolean per le azioni possibili; le uniche operazioni che si possono effettuare su di esse sono ottenerne il valore e cambiarlo. Questi valori vengono usati nelle varie pagine per effettuare i controlli sulle autorizzazioni. Ma questi, essendo ripetuti in ogni pagina, sono a volte incoerenti o scritti in maniera errata, causando i problemi menzionati.

Capitolo 4

Soluzioni

4.1 Presentazione

Per prima cosa ho risolto i problemi grafici presenti nelle varie pagine. In particolare ho utilizzato gli elementi di Bootstrap secondo lo schema seguente: i pannelli vengono utilizzati per rappresentare un elemento nella sua completezza, ad esempio una Scheda Informativa, un Reperto o un Contenuto Digitale; all'interno di un pannello i dati sono strutturati utilizzando le classi helper css container, row e column: il container contiene una row per ogni dato, ogni row contiene due colonne, una per il nome del dato e una per il valore. Le tabelle usano il plugin di Bootstrap Datatables, che gli da uno stile grafico piacevole e la possibilità di ordinare le righe in base al valore di una qualsiasi colonna e di cercare all'interno di essa. Inoltre è stato cambiato il modo in cui i dati venivano presentati all'interno delle tabelle: per esempio, nel caso della tabella che mostra l'elenco dei Reperti di una Collezione, le pagine ad essi relative erano accessibili tramite un iconcina posizionata in fondo alla riga, rendendone poco chiaro il significato; similmente venivano adottate delle iconcine per la modifica o la cancellazione del Reperto e per l'aggiunta di Contenuti Digitali. Ora i Reperti sono accessibili tramite un collegamento ipertestuale con il nome del Reperto come testo; le altre azioni sono invece disponibili come un gruppo di bottoni sull'ultima colonna.

Delle pagine incomplete dal punto di vista grafico erano quelle in cui si possono definire i sistemi di Catalogazione e di Collocazione; la form contenuta in ciascuna pagina presentava tutti gli elementi in maniera disordinata. Inoltre qualora venisse scelto, tra i vari tipi possibili per un attributo del sistema di Catalogazione, il tipo Enumerazione venivano aggiunti venti coppie di campi di testo uno dopo l'altro rendendo molto confusa la pagina. In generale non era poi chiaro dove iniziassero e dove finissero i campi d'input

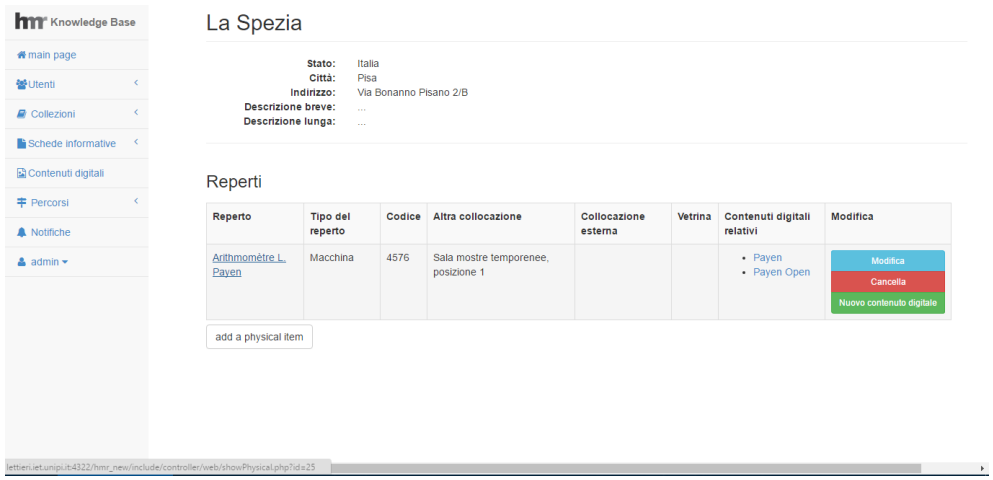

Figura 4.1: Tabella dei Reperti di una Collezione

relativi ad un attributo. Pertanto ho strutturato la pagina come segue: ogni attributo è contenuto all'interno di un pannello retrattile. In questo modo si possono aprire soltanto gli attributi che si è interessati a modificare. All'interno di ogni pannello ci sono i campi relativi a tale attributo. Soltanto nel caso in cui si specifichi il tipo Enumerazione per l'attributo, viene mostrato un pannello retrattile che se cliccato mostra le venti coppie valore - descrizione. Se un attributo era già stato specificato in precedenza il titolo del pannello riporta il nome dell'attributo e il tipo, in modo che l'Utente sappia già di che attributo si tratta senza doverlo aprire per forza.

4.2 Codice

Analizziamo per primo il caso della procedura di assegnamento dei responsabili; quando veniva chiamato attraverso la pagina di modifica Utente, l'assegnamento era effettuato da due metodi della classe Utente. La form della pagina permette di selezionare le collezioni per le quali un Utente dev'essere Responsabile o Catalogatore, perciò i due metodi della classe utente provvedevano uno a rimuovere tutte le relazioni tra l'Utente in questione e qualsiasi collezione, l'altro a inserire tutte quelle specificate nella form. Queste operazioni venivano fatte attraverso query dirette al database, anziché utilizzare il metodo setManager della Collezione; a quest'ultima mancava anche un metodo removeManager, che ho aggiunto. Esistevano già i metodi insertCataloguers e removeCataloguers, che accettano un array contenenti gli id degli

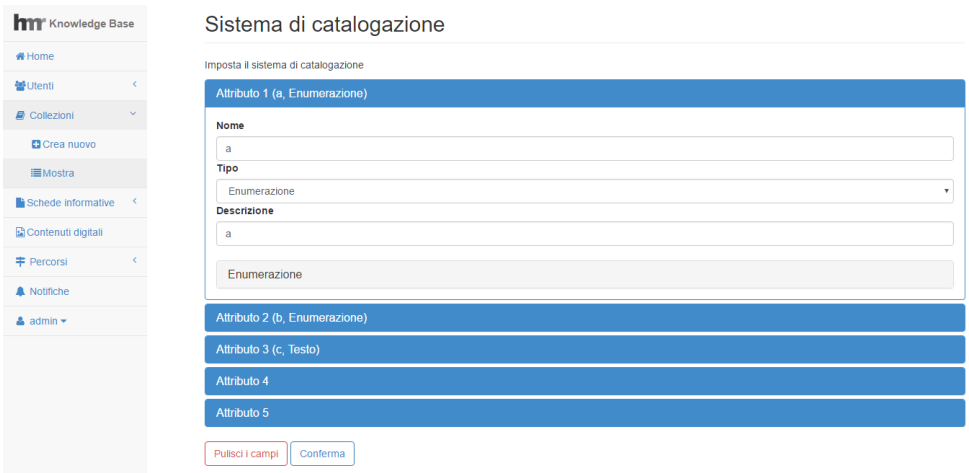

Figura 4.2: Form definizione sistema di Catalogazione

Utenti interessati. La procedura di assegnamento quindi consiste ora in un semplice algoritmo: nel caso del ruolo di Catalogatore, si considera l'insieme A, composto dalle collezioni nelle quali l'Utente ha il ruolo di Catalogatore, contenuto nella proprietà cataloguer_at_ dell'Utente, mentre l'insieme B è composto dalle Collezioni selezionate dall'admin dalla form. A questo punto le Collezioni da cui rimuovere l'Utente come Catalogatore sono quelle dell'insieme B - A, quelli da aggiungere sono dell'insieme A - B. Si possono quindi chiamare i metodi removeCataloguers e insertCataloguers su ciascuna di esse, passando come valore l'id dell'Utente. Lo stesso ragionamento può essere seguito per le Collezioni da cui rimuovere il ruolo di Responsabile. Di seguito viene riportato il codice per il ruolo di Catalogatore:

```
$cataloguersToRemove = array_diff($this->cataloguer_at_,
        $info['cataloguer_at']);
$cataloguersToAdd = array_diff($info['cataloguer_at'],
        $this->cataloguer_at_);


foreach($cataloguersToRemove as $collectionId)
{
        $collection = Collection::fetch($this->db_,
            $collectionId);

        $collection->removeCataloguers(array($this->id_));
```

```
}


foreach($cataloguersToAdd as $collectionId)
{
        $collection = Collection::fetch($this->db_,
            $collectionId);

        $collection->insertCataloguers(array($this->id_));
}
```

L'altro problema evidente del codice erano i nomi poco chiari in alcune parti, soprattutto quella dei percorsi. Un esempio esplicativo è quello dei metodi della classe InfoVersion listInfos1, listInfos2, listInfos3 e listInfos4 che malgrado fossero distinti da un solo numero, svolgevano compiti molto diversi. listInfos1 si occupava di restituire gli id delle Schede Informative a cui si riferisce la Scheda Informativa chiamante; listInfos2 restituiva gli id delle Schede Informative che riferiscono la Scheda Informativa chiamante; listInfos3 restituiva gli id dei Reperti che riferivano la Scheda Informativa chiamante; listInfos4 restituiva gli id dei Percorsi che contengono la Scheda Informativa chiamante. I nomi sono stati sostituiti come segue: listInfos1 => getRelatedInfos, listInfos2 => getRelatedByInfos, listInfos3 => getRelatedPhysicals, listInfos4 => getRelatedRoutes.

Una cosa che si può notare osservando il codice è che nonstante esistano classi per descrivere tutti gli elementi del dominio dell'applicazione, tutte le funzioni si passano tra loro esclusivamente gli identificativi degli oggetti, delegando alla funzione chiamata il compito di estrarre dal database l'oggetto stesso. Questo comportamento non è a mio avviso il migliore, in quanto lo stesso oggetto viene richiesto più volte dal database e viene salvato in più copie nella memoria, ma provare ad affrontare il problema per correggerlo avrebbe comportato un refactoring dell'intero codice, un compito troppo oneroso in termini di tempo.

4.3 Estensione

4.3.1 Menu

Per introdurre un Menu che tenga traccia della pagina in cui si trova l'Utente è stato usato l'idea seguente: ogni pagina chiama il metodo statico View::header(...) che accetta diversi parametri utilizzati per inserire file JavaScript o CSS aggiuntivi. È stato introdotto un nuovo parametro $current-

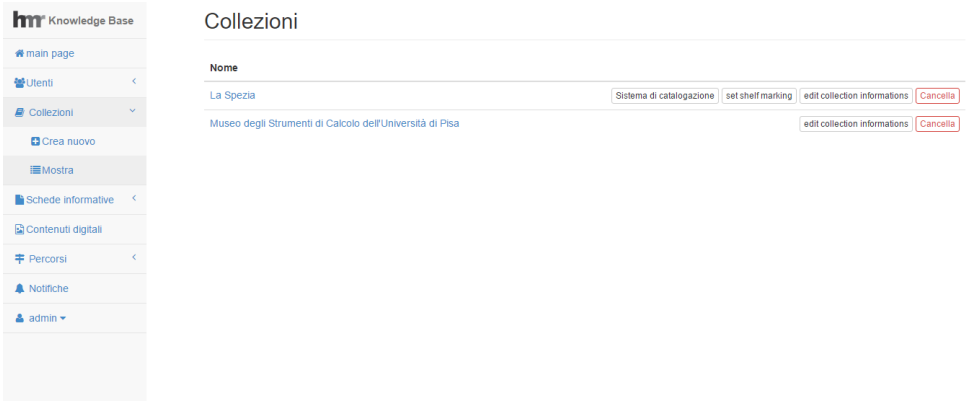

Figura 4.3: Menu laterale con pagina di visualizzazione delle Collezioni

Page che indica il nome del link del menu che dev'essere attivo quando si visita la pagina. Questo parametro viene usato per impostare la proprietà omonima della classe NavBar in modo che ne tenga conto durante la generazione del codice HTML tramite il metodo display(). All'interno del Menu i link sono divisi in più gruppi come Utenti, Schede Informative, Collezioni, Percorsi etc. Questi gruppi quando vengono cliccati si aprono mostrando i link al loro interno. Le classi Bootstrap che regolano la presentazione e lo stato del menu sono: "collapse in" per aprire un gruppo, "active" per far cambiare il colore di sfondo in modo da rendere il link riconoscibile come attivo. Per non ripetere molte volte lo stesso controllo sono stati introdotti quattro metodi:

```
public function isCurrentPage($page)
{
        return $this->currentPage == $page;
}

public function printCollapseIn($pages)
{
        foreach($pages as $page)
        {
                if($this->isCurrentPage($page))
                        return 'collapse in';
        }
}
```

```
public function printClassActive($page)
{
        if($this->isCurrentPage($page))
                return 'class="active"';
}

public function printClassActiveList($pages)
{
        foreach($pages as $page)
        {
                if($this->isCurrentPage($page))
                        return 'class="active"';
        }

        return '';
}
```

isCurrentPage($page) restituisce vero o falso a seconda che $page sia la pagina visitata al momento. printCollapseIn($pages) restituisce la stringa "collapse in" se una qualsiasi tra le pagine nell'array $pages corrisponde alla pagina corrente; questo metodo è utilizzato per tenere il gruppo aperto se la pagina corrente appartiene ad esso. printClassActive($page) e printClassActive($pages) fanno la stessa cosa, ma la seconda lo fa su un array di stringhe anziché su una stringa. Il primo serve per assegnare la classe "active" ad un elemento, la seconda per assegnarla ad un gruppo. Un esempio di utilizzo, estratto dal metodo view(), è:

```
<?php
$pages = array('accept_reject', 'show_users');

if(!is_null($this->user) && $this->user->canAcceptUser()) {
    ?>
        <li <?= $this->printClassActiveList($pages) ?>>
                <a  href="#">
                        <i class="fa fa-users fa-fw"></i>
                                <?php echo Language::get('
                                    menu_manage_users'); ?>
                        <span class="fa arrow"></span>
                </a>
        <ul class="nav nav-second-level <?= $this->
            printCollapseIn($pages) ?>">
        <!-- Accept users -->
        <li>
        <a href="<?php echo WEB_URL.'acceptUsers.php';?>"
```

```
                <?= $this->printClassActive($pages[0]) ?>>
                <i class="fa fa-plus-square fa-fw"></i><?
                    php echo Language::get('action_accept');
                     ?>
        </a>
        </li>

        <!-- Show users -->
        <li>
        <a href="<?php echo WEB_URL.'showUsers.php';?>"
        <?= $this->printClassActive($pages[1]) ?>>
        <i class="fa fa-list fa-fw"></i><?php echo Language
            ::get('action_show');?>
        </a>
        </li>
        </ul>
        <!-- /.nav-second-level -->
        </li>
<?php } ?>
```

L'array $pages contiene i nomi delle pagine raggiungibili dai link del gruppo, ed è quindi utilizzato per determinare se il gruppo debba essere aperto o chiuso; ognuno degli elementi dell'array viene poi utilizzato per determinare quale sia il link attivo. Nella pagina acceptUsers.php verrà chiamato View::header.php($user, ..., 'accept_reject'), che imposterà questa stringa come $currentPage e mostrerà il menu laterale con il link a acceptUser come attivo.

4.3.2 Cronologia Attività

Nel database venivano già salvati i dati necessari all'introduzione di questa funzionalità. Infatti le versioni passate venivano mantenute completamente, ma erano visibili solo dal revisore al momento dell'assegnamento di una di esse in redazione. Ho quindi creato un nuovo metodo della classe Utente, che va a recuperare le Versioni di Scheda Informativa per la quale risulta creatore, redattore o revisore. Il metodo è stato chiamato getInfoEditHistory(), e il codice è il seguente:

```
public function
getInfoEditHistory()
{
        $this->db_->query("SELECT id, version
                FROM info_version
```

```
                WHERE author = '{$this->id_}'
                OR reviewer = '{$this->id_}'
                ORDER BY version DESC");

        $infoVersionIds = array();
        $infos = array();

        while($row = $this->db_->fetchResult())
        {
                $infoVersionIds[] = $row;
        }

        foreach($infoVersionIds as $infoVersion)
        {
                $info = Info::fetch($this->db_,
                    $infoVersion['id']);

                $version = $this->db_->toDateTime(
                    $infoVersion['version']);

                $id = array('info' => $info, 'version' =>
                    $version);

                $infos[] = InfoVersion::fetch($this->db_,
                    $id);
        }

        return $infos;
}
```

Attraverso la query SQL vengono prese tutte le versioni in cui l'Utente appare come revisore o come redattore; i tre casi possibili sono:

- l'utente è revisore e redattore è null, allora l'utente è il creatore della scheda
- l'utente è revisore e il redattore non è null, allora l'utente è il revisore della scheda
- l'utente non è revisore ma è redattore, allora l'utente è il redattore della scheda.

Viene quindi restituito un array composto da elementi della classe InfoVersion.

Questo metodo viene utilizzato nella pagina showUserHistory.php attraverso il metodo statico della classe InfoView showVersionHistory(*user*,db) che prende in ingresso l'Utente che richiede la Cronologia e un istanza di connessione al Database. Il codice è riportato di seguito:

```
$versions = $user->getInfoEditHistory();

foreach($versions as $version)
{
        $authorId = $version->getAuthor();
        $reviewerId = $version->getCreatorReviewer();
        $reviewer = User::fetch($db, $reviewerId);

        if($reviewerId == $user->getUsername())
        {
                if(is_null($authorId))
                {
                        $activity = 'history_create';
                        $dateString = 'field_created_on';
                }
                else
                {
                        $activity = 'history_review';
                        $dateString = 'field_assigned_on';
                }
        }
        else //user is author
        {
                $activity = 'history_author';
                $dateString = 'field_assigned_on';
        }



        if(!is_null($authorId))
        {
                $author = User::fetch($db, $authorId);
        }
        else
        {
                $author = NULL;
        }
```

```
/*      codice HTML omesso
 *      che presenta l'elenco delle versioni
 *      nello stile a pannelli retrattili
 *      con visibile la data a cui risale
 *      la versione
 */
```

Le prime righe di codice servono per determinare il ruolo dell'Utente per quella versione, secondo la logica descritta pocanzi. Nel codice HTML si usano le variabili $activity e $dateString per scegliere i messaggi che riassumono l'attività, ad esempio "Creata in data:" o "Revisionata da:". Infine viene usato il metodo statico InfoView::showVersion($version, ...) per mostrare la Versione della Scheda Informativa in questione.

4.3.3 Commenti

L'introduzione dei Commenti è stata anch'essa semplice e circoscritta all'aggiunta di una colonna alla tabella info_version per contenere il testo del commento. È stato aggiunto un campo alla form che permette al revisore di accettare o rifiutare una redazione specificando delle motivazioni attraverso un campo di testo. Questo commento viene poi inserito permanentemente nella versione e sarà visibile a tutti i Revisori. Le modifiche al codice sono state ristrette al cambiamento della classe InfoVersion con l'aggiunta di un parametro al costruttore per passare la stringa commento e all'aggiunta della proprietà comment_, i metodi getComment() e setComment() per ottenere e modificare il valore della stringa, e all'adattamento dei metodi statici create(...) e fetch(...) per integrare la proprietà aggiuntiva.

Anche per le Proposte sono stati aggiunti i commenti, ma a differenza delle Versioni, non è stata aggiunta una colonna addizionale. Questa scelta è stata fatta perché le Proposte, quando vengono rifiutate sono semplicemente cancellate dal database, quando vengono accettate vengono mantenute ma con una relazione diretta verso la Scheda Informativa così generata. È stato deciso di seguire dunque la seguente politica: se una Proposta viene accettata, il commento viene inserito direttamente nella Scheda Informativa appena creata, comparendo dunque come commento nella Versione relativa alla creazione; se una Proposta viene rifiutata, il commento viene usato solamente per la creazione delle Notifica (secondo i metodi spiegati nella prossima sezione) e poi viene scartato insieme al resto della Proposta.

4.3.4 Notifiche

Le notifiche hanno richiesto la creazione di una nuova tabella, chiamata Notice poiché il nome Notification veniva già usato nel codice preesistente con un altro significato, e l'aggiunta di una colonna alla tabella registered_user dal nome last_notice_check utilizzato per memorizzare il timestamp dell'ultima volta che l'utente ha controllato le proprie notifiche. La tabella Notice ha quattro colonne:

id : è l'id della notifica, è un intero diverso da zero ed è unico;

user_id : è l'id dell'utente a cui è indirizzata la notifica, è una stringa di massimo 255 caratteri;

date : è il timestamp del momento in cui è stata generata la notifica;

text : è il testo della notifica, è una stringa di massimo 255 caratteri.

La classe Notice ha cinque coppie di costanti, una coppia per tipo di notifica possibile:

Revisione : una costante indica una revisione accettata, l'altra una rifiutata;

Proposta : una costante indica una proposta accettata, l'altra una rifiutata;

Capability : una costante indica una capability concessa, l'altra una negata;

Responsabile : una costante indica l'assegnamento del ruolo di responsabile, l'altro la revoca;

Catalogatore : una costante indica l'assegnamento del ruolo di catalogatore, l'altro la revoca;

Per creare una notifica, una qualsiasi altra classe può utilizzare il metodo statico Notice::create($db, $user, $noticeType, $noticeObject), che si occupa di chiamare uno dei metodi privati predisposti alla generazione del testo della notifica in base alla variabile $noticeType. L'oggetto $noticeObject è di tipo diverso a seconda del valore di $noticeType.

È una Versione nel caso di una notifica di tipo Revisione, una Collezione nel caso del tipo Responsabile o Catalogatore, una stringa col nome della capability nel caso del tipo Capability o una mappa ('proposal' => InfoProposal, 'comment' => string) nel caso del tipo Proposta. Utilizzando questi

oggetti viene costruito un testo adeguato al tipo di notifica, che viene poi inserita nel database.

La classe Utente è stata estesa con due metodi: getNotices($limit, $update) e getNumberUnseenNotices(). La prima estrae dal database un numero pari a $limit di notifiche, e controlla che il timestamp della Notifica sia minore del timestamp dell'ultimo controllo dell'utente, nel qual caso marca la notifica come già vista. Se $update ha valore true, viene aggiornato il timestamp dell'utente; il motivo per cui non viene fatto a prescindere sarà chiaro analizzando il secondo metodo. Dopodiché restituisce un array con tutte le notifiche richieste. Di seguito viene riportato il codice:

```
public function
getNotices($limit, $update)
{
        $notices = array();

        $this->db_->query("
                SELECT n.*, n.date < u.last_notice_check AS
                    seen
                FROM notice n INNER JOIN registered_user u
                ON n.user_id = u.username
                WHERE n.user_id = '{$this->id_}'
                ORDER BY n.date DESC " .
                ( is_null($limit) ? "" : "LIMIT {$limit}" )
                    );


        $rows = array();
        while($row = $this->db_->fetchResult())
        {
                $rows[] = $row;
        }



        foreach($rows as $row)
        {
                $user = User::fetch($this->db_, $row['
                    user_id']);
                $notices[] = new Notice($this->db_,
                        $row['id'], $user, $row['text'],
                        $row['date'], $row['seen'] == 't');
        }
```

```
        if($update)
        {
                $this->db_->query("
                        UPDATE registered_user
                        SET last_notice_check = now()
                        WHERE username = '{$this->id_}'");
        }

        return $notices;
}
```

Il metodo getNumberUnseenNotices() serve invece per sapere il numero di notifiche non ancora viste dall'utente. Questo metodo è stato pensato appositamente per rendere immediatamente disponibile questa informazione nel menu laterale. Infatti, oltre al link che permette di accedere all'elenco completo delle notifiche, nel caso in cui siano presenti una o più notifiche non lette, comparirà un cerchietto con scritto il numero di notifiche non lette. Il codice è abbastanza semplice e riutilizza la funzione descritta precedentemente:

```
public function
getNumberUnseenNotices()
{
        $notices = $this->getNotices(NULL, false);

        $unseenNotices = array_filter($notices,
                function($var)
                {
                        return !$var->isSeen();
                }
        );

        return count($unseenNotices);
}
```

Si usa la funzione array_filter per estrarre dalle notifiche soltanto quelle non viste, dopodiché si restituisce il numero di elementi nell'array filtrato.

Le notifiche sono state organizzate graficamente secondo il seguente schema; ogni notifica è contenuta in un pannello retrattile, la cui barra del titolo mostra i primi 50 caratteri. In questo modo l'utente può avere una vaga idea di quello che sarà il messaggio completo della notifica e non deve scorrerle una alla volta. Inoltre viene mostrata un'indicazione temporale espressa nel formato "x minuti/ore/giorni fa", rendendo più immediato quanto sia recente o meno la notifica. Una volta cliccato sull'anteprima della notifica, il pannel-

lo si apre mostrando il testo completo. Qualora la notifica non fosse ancora stata vista, essa sarà colorata di un rosso chiaro per evidenziarla rispetto alle altre.

4.4 Capabilities

Sono state analizzate due possibili soluzioni per risolvere i problemi relativi alle capabilities: la prima prevede un refactoring completo per centralizzare tutte le operazioni di verifica dei permessi all'interno della classe Utente, anziché riutilizzare lo stesso codice in ogni pagina col rischio di incoerenze; la seconda prevede la ricerca di tutte le pagine che mostravano problemi di questo tipo e la correzione del codice presente in essa. Nonostante la prima portasse ad una migliore struttura del codice, una maggiore manutenibilità e più sicurezza da errori di incoerenza nelle decisioni sulle autorizzazioni, avrebbe richiesto una quantità di tempo molto maggiore rispetto al tempo allocato per questo progetto. Si è optato dunque per la seconda soluzione, che poteva essere più facilmente applicabile navigando il sito seguendo diversi casi di utilizzo per scoprire eventuali malfunzionamenti. Diversi errori di questo tipo sono stati corretti e le capabilities vengono applicate correttamente sia nel caso dell'autorizzazione sia nel caso del divieto.

Tuttavia quella adottata è una soluzione che dovrebbe essere temporanea; il suggerimento lasciato per futuri sviluppi ed estensioni del progetto è di ricorrere al refactoring il prima possibile, in modo da rendere più facili eventuali cambiamenti della logica di accesso a determinati contenuti.

Capitolo 5

Risultati

I risultati ottenuti sono più che soddisfacenti: il sito è passato da uno stadio in cui era pressocché inutilizzabile ad uno in cui è utilizzabile produttivamente per lo scopo per cui è stato realizzato. L'unica funzionalità che, pur operando correttamente, potrebbe essere migliorata ampiamente su vari aspetti è quella dei percorsi. Infatti il codice relativo presenta una struttura disordinata e con uno stile non coerente con il resto del software. L'aspetto grafico inoltre non rende ben chiaro lo spostamento all'interno del percorso, e le informazioni sono sparpagliate per la pagina, dando all'utente un esperienza dispersiva; nel capitolo successivo si parlerà di come si potrebbe migliorare la funzionalità. Fatta eccezione per i Percorsi, il resto del sito funziona soddisfacendo le specifiche ed è utilizzabile per svolgere completamente al suo interno il lavoro di catalogazione universale che il progetto CHKB proponeva. Con l'aggiunta dei Commenti e delle Notifiche si è eliminata la necessità di comunicazione esterna al sito; con la Cronologia Attività si è eliminato il rischio che un utente perdesse traccia delle modifiche da lui effettuate.

Capitolo 6

Possibili sviluppi futuri

La pagina di esplorazione di un percorso può essere migliorata, soprattutto dal punto di vista grafico, per rendere più piacevole e chiara la navigazione. Una possibile idea, che non è stata realizzata per mancanza di tempo, è sfruttare le animazioni offerte da jQuery (compreso in Bootstrap) insieme alla tecnologia AJAX per creare una singola pagina che attraverso effetti di transizione rappresenta lo spostamento tra le tappe del percorso, o tra percorsi correlati.

Una seconda possibilità, più semplice rispetto alla prima, è una pagina in cui compaiano le anteprime di tutti gli elementi del percorso disposte orizzontalmente e collegate tra di loro da linee che rappresentano l'ordine in cui visitare l'itinerario; in verticale comparirebbero gli elementi dei percorsi correlati, e cliccando su di essi la pagina verrebbe ricaricata con gli elementi del nuovo percorso.

Un'altra funzionalità che potrebbe essere utile è un sistema di messaggi privati tra Utenti. Questo aiuterebbe sotto diversi aspetti: i revisori potrebbero mettersi d'accordo per dividersi le Schede da revisionare a seconda di tempo, interessi e conoscenze, i Redattori potrebbero contattare i Revisori in caso di dubbi o per segnalare dei problemi. L'admin sarebbe inoltre rintracciabile attraverso il sito, invece che esclusivamente tramite email, per richiedere l'aggiunta di capabilities: ad esempio un utente, precedentemente registrato come utente semplice, potrebbe decidere di mettere a disposizione la propria collezione privata; potrebbe quindi scrivere un messaggio privato all'admin che esaminata la richiesta (eventualmente dopo uno scambio di messaggi) può decidere se acconsentire o meno.

Infine, come accennato nei capitoli precedenti, bisognerebbe effettuare un refactoring di molte parti di codice. È importante che si sposti nella sua interezza il controllo delle autorizzazioni all'interno della classe Utente, in modo da rendere il codice semplice e sicuro da incoerenze nel calcolo delle capa-

bilities. È altrettanto importante cambiare molte funzioni affinché accettino non esclusivamente gli identificativi degli oggetti, ma gli oggetti stessi. Questo è vero perché in questo modo si hanno molte meno query al database per recuperare gli oggetti che probabilmente esistono già in memoria, si possono passare gli oggetti per riferimento evitando le operazioni di copia e si impedisce che lo stesso oggetto sia presente più volte ma con valori diversi per le sue proprietà: questo è possibile perché se la funzione chiamata crea per sé un'altra copia dell'oggetto ed opera modifiche su di esso, queste si rifletteranno sul database, ma non sulla copia della funzione chiamante, che potrebbe ritrovarsi con dati inconsistenti. Il codice esistente non risente di questo problema perché quasi mai l'oggetto il cui identificativo viene usato come parametro viene riutilizzato dalla funzione chiamante, ma un codice futuro potrebbe essere limitato da questa problematica.

Bibliografia

[1] Progetto e realizzazione di CHKB, una base di conoscenza web per il Museo degli Strumenti per il Calcolo.
https://etd.adm.unipi.it/t/etd-11132012-133307/

[2] Progettazione e Realizzazione di un Modulo per i Percorsi Virtuali in una Base di Conoscenza Informatica.
http://hmr.di.unipi.it/Documenti/HMR_2015d_DA-TirCHKB.pdf