UNIVERSITÀ DEGLI STUDI DI TRENTO

Microbit

Software per la gestione di micro-pagamenti in Bitcoin

Martino Salvetti, 149722

Web Architectures
professore Marco Ronchetti

Indice generale

Introduzione

Questo progetto prevede la realizzazione di una applicazione web mediante EJB 3.1, JSP, il layer di persistenza Hibernate e il DBMS MySQL. È stato utilizzato l'ambiente di sviluppo Eclipse Juno con il relativo plugin Web Tool Platform (WTP) e il client ufficiale del progetto Bitcoin.

L'applicazione realizzata si occupa di gestire dei depositi in valuta Bitcoin degli utenti e i relativi trasferimenti fra gli utenti stessi e con tutta la rete Bitcoin.

L'applicazione tiene traccia delle transazioni interne (fra gli utenti) ed esterne al server (attraverso la rete Bitcoin) e permette agli utenti di visualizzare le transazioni in cui sono direttamente coinvolti.

Bitcoin è una moneta elettronica P2P nata nel 2009; essa è la prima implementazione del concetto di cybermoneta descritto nel 1998 da Wei Dai [1]; esso è basato sull'idea dell'uso della crittografia per controllare la creazione e il trasferimento di moneta, invece di appoggiarsi ad autorità centrali. Ad oggi, un Bitcoin (BTC) è valutato 14€ [2], il valore è determinato unicamente dalle offerte di altre valute e beni e servizi in cambio di Bitcoin, cioè esso segue il libero mercato.

Obiettivi e descrizione del progetto

La necessità di realizzare un sito per agevolare il pagamento di piccoli importi in Bitcoin è dettata da alcune caratteristiche delle rete P2P che la rendono non adatta un utilizzo di questo tipo.

Bitcoin prevede un costo di minimo per transazioni [3] con importo inferiore a una quota (0.01BTC per transazioni inferiori a 0.01BTC, per le altre ci sono regole più complesse); inoltre, le transazioni richiedono in media 10 minuti per venir confermate (cioè per venire incluse in un blocco[1]).

Si potrebbe osservare che progettare un'applicazione centralizzata per supportare un'architettura P2P comporta la perdita dei molti vantaggi che suddetta architettura ha introdotto[2]. Allo scopo di attenuare gli svantaggi introdotti dall'architettura centralizzata, si e deciso di aggiungere un ulteriore identificativo agli utenti di Microbit: il loro indirizzo Bitcoin associato. In questo modo, nel caso in cui l'applicazione centralizzata non dovesse essere disponibile per un malfunzionamento o per sovraccarico, i pagamenti non verrebbero interrotti ma potrebbero essere effettuati direttamente mediante Bitcoin. In questa situazione, si perderebbero i vantaggi forniti dall'applicazione web, ma si potrebbe comunque continuare ad operare.

Analisi dei requisiti

In questo capitolo vengono definiti e motivati i requisiti dell'applicazione che andranno a delineare le funzioni progettate e implementate.

Descrizione

Si è partiti analizzando ad alto livello l'applicazione web da realizzare; la caratteristica principale riguarda la possibilità di effettuare dei trasferimenti di valuta (Bitcoin) fra gli utenti del sito.

1 Le transazioni sono memorizzate in una catena di blocchi globale. Ogni blocco include una Proof of Work, la cui difficoltà viene calcolata dinamicamente in base alla potenza di calcolo dell'intera rete Bitcoin; in questo modo viene regolata la frequenza dei blocchi e si garantisce la sicurezza della rete.

2 I vantaggi apportati da un'architettura P2P comprendono l'affidabilità e la resilienza, per quanto riguarda la disponibilità del servizio; e l'indipendenza da un'autorità centrale e l'impossibilità di produrre in modo incontrollato valuta, per quanto riguarda la caratteristiche dal punto di vista economico.

Attori – utenti del sistema

È stato identificato un unico attore: l'utente. Non è previsto invece nessun amministratore o moderatore: l'agire degli utenti non deve essere soggetto a verifica; ogni transazione è non invertibile e il sito non si occupa di risolvere le controversie fra gli utenti[3]. Queste caratteristiche derivano dal design della rete Bitcoin. Si è scelto di mantenere il più possibile continuità con le caratteristiche di tale rete P2P su cui il sito è basato.

Entità del sistema

Le entità con cui l'utente può interagire sono:

- **transazioni**: lo scopo dell'applicazione è la creazione di transazioni per permettere il trasferimento di valuta verso gli altri utenti;
- **rubrica**: semplifica il compito dell'utente diminuendo la probabilità di errore;
- **history**: l'utente potrà consultare una lista delle precedenti transazioni effettuate e ricevute.

Azioni possibili

L'azione principale per cui il sito viene progettato è la possibilità di effettuare transazioni – micro-transazioni – fra gli utenti del sito.

Si nota subito la necessità di un meccanismo per accreditare e prelevare valuta dal sito web: questo si traduce nella possibilità di effettuare dei versamenti a un qualsiasi utente della rete Bitcoin e nel fornire un indirizzo all'utente su cui effettuare i versamenti.

Per supportare al meglio queste funzionalità verrà progettata una rubrica; utilizzata sia per gli indirizzi interni al sito (id degli utenti), sia per gli indirizzi Bitcoin (per effettuare prelievi).

Per supportare l'utente nell'utilizzo della valuta Bitcoin, e considerata l'alta instabilità del loro valore, l'applicazione permetterà di consultare in modo veloce il valore di cambio con altre valute (Euro e Dollaro). Esso sarà ottenuto, in tempo reale, da Mtgox [2]: portale che gestisce l'80% della compravendita mondiale di Bitcoin.

Requisiti funzionali

I requisiti funzionali derivano direttamente dalle azioni che gli utenti dovranno poter effettuare:

1. l'invio dei pagamenti verso utenti dell'applicazione web e verso utenti esterni della rete Bitcoin;
2. fornire una rubrica che permetta di assegnare un nome a un indirizzo Bitcoin oppure a un id Microbit;
3. visualizzazione dei pagamenti effettuati e degli accrediti ricevuti;
4. Gestione del login e del logout, protetto mediante password;
5. consultazione del valore della valuta Bitcoin.

Requisiti non funzionali

Prendendo in considerazione il tipo di applicazione è evidente come la (1) **sicurezza** ricopra una

3 Servizi come Paypal, permettono invece di interferire con le transazioni degli utenti. Questo avviene per prevenire o sventare le truffe.

particolare centralità in Microbit; un bug potrebbe permettere la sottrazione di Bitcoin da parte degli utenti stessi, oppure un accesso non autorizzato al server o la sottrazione di una password potrebbe permettere a terzi di effettuare le stesse irregolarità.

Un altro requisito riguarda la (2) **riservatezza**; verranno memorizzati solo i dati indispensabili. Nei limiti del possibile, non dovrebbe essere possibile risalire all'identità di un utente utilizzando i dati forniti da Microbit, (ad esclusione dei casi previsti dalla legge).

Progettazione

Use Case Diagram

Il diagramma dei casi d'uso rappresenta le azioni che gli attori del sistema possono compiere. In questo caso, come anticipato nel capitolo precedente, esiste un unico attore.

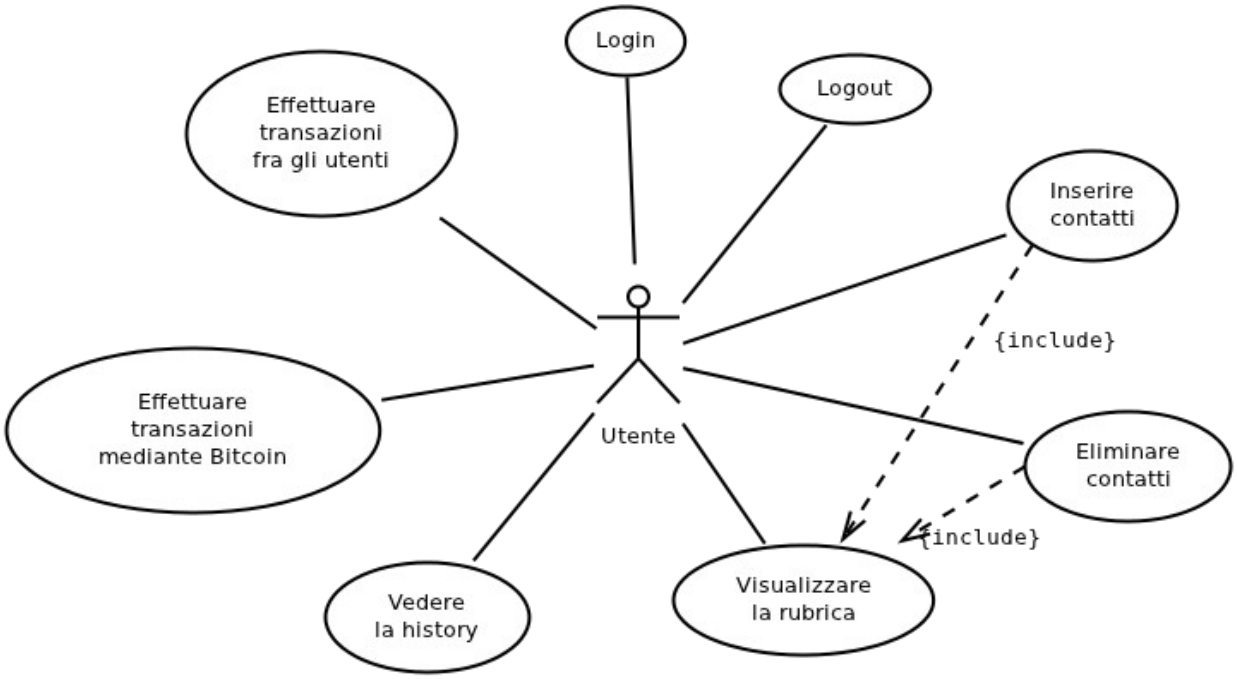


***Immagine 1**: Use case diagram*

Architettura

Il progetto è stato sviluppato seguendo una precisa divisione in livelli logici. Questa organizzazione presenta vari vantaggi fra cui la manutenibilità del codice e l'indipendenza fra i livelli (fra l'interfaccia e la *business logic*, o fra la business logic e il DBMS).

I livelli in cui l'applicazione è suddivisa sono:

- l'interfaccia web, implementata mediante delle pagine JSP interfacciate a delle classi JavaBean;
- la *business logic*, implementata usando la tecnologia EJB 3.1;
- il livello di persistenza, implementato attraverso Hibernate e il DBMS MySQL.

Schema del Database

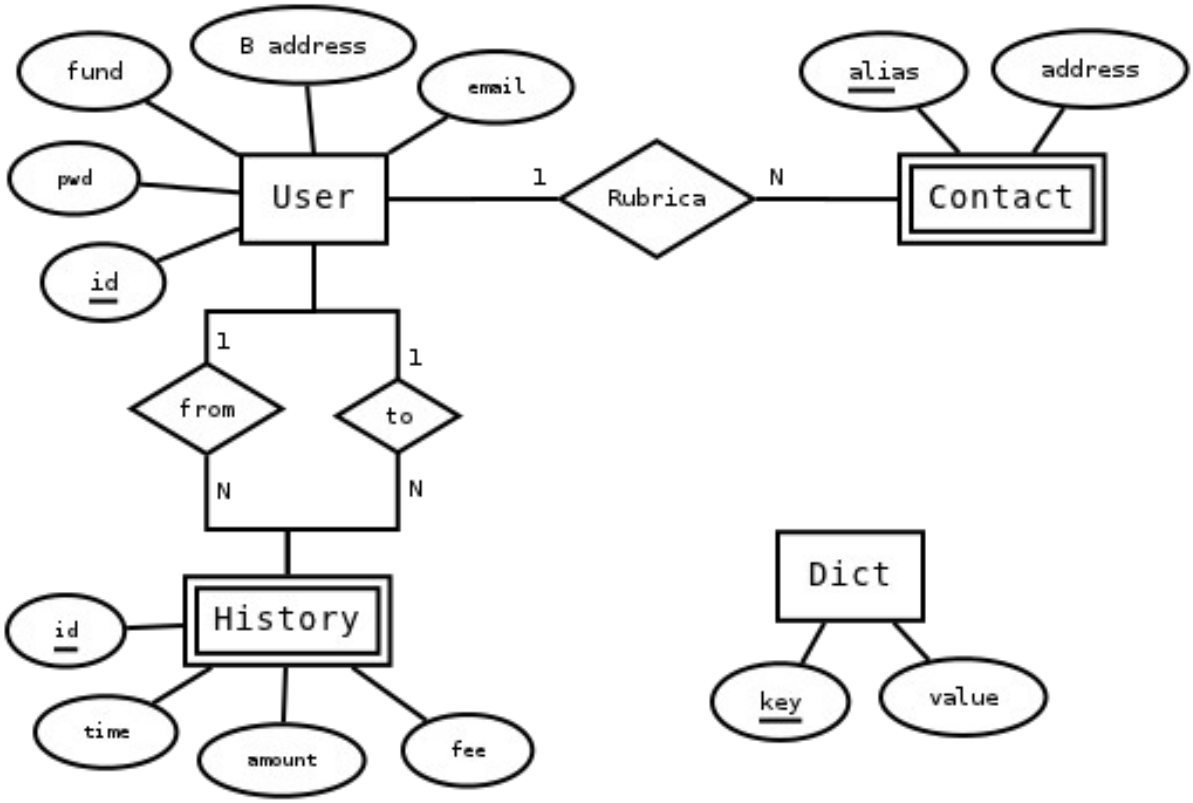


***Immagine 2**: diagramma ER*

Sono state identificate 3 entità logiche che verranno gestite dal livello di persistenza:

- **User**: l'utente registrato. Il sito gestirà le sessioni degli utenti, necessita quindi di un record per ogni utente registrato all'interno del database. Come si può notare dall'immagine 2, gli utenti non sono identificati da un *username*, ma sono associato ad un *id* numerico. Questa scelta risponde a entrambi i requisiti non funzionali: aumenta il livello di sicurezza, perché rende più difficile utilizzare le coppie username e password ottenute illecitamente da altri siti su Microbit, (è un comune comportamento utilizzare lo stesso username e la stessa password su molti siti diversi, creando un problema di sicurezza); riguardo il secondo requisito non funzionale, la mancanza un username rende più difficile risalire all'identità di un utente del sito senza il suo consenso. Il campo *B address*, o *deposit_address*, rappresenta l'indirizzo Bitcoin su cui l'utente deve effettuare i versamenti per aggiungere fondi al proprio account. Data l'unicità dell'indirizzo e la sua reperibilità (dopo aver effettuato un versamento esso rimane nello storico delle transazioni) si è scelto di utilizzarlo – in sostituzione all'id numerico – per identificare l'utente univocamente, sia in fase di login sia per quanto riguarda le transazioni.
- **Contact**: contiene le rubriche. È un'entità debole perché id della tabella *User* è parte della chiave. Per ogni utente può memorizzare un *set* di contatti, cioè coppie *alias*, *address*. Per address si intende sia un indirizzo Bitcoin (di un utente di Microbit, oppure un indirizzo esterno), sia un id numerico associato a un utente.
- **History**: le transazioni avvenute. Sono divise in *effettuate* e *ricevute*. Identificate da un id

memorizzano un timestamp, la valuta trasferita e la commissione sostenuta.

- **Dict**: è una tabella utilizzata come HashMap generica (**key**: VARCHAR(20); **value**: VARCHAR(100)). Essa viene utilizzata dal livello *business logic* per l'immagazzinamento arbitrario di dati sotto forma di stringhe.

Nel file **create_database.sql.txt**, presente nella cartella del progetto, è riportato il codice SQL necessario per creare lo schema del database.

Progettazione delle classi

La progettazione della classi è stata suddivisa in più passaggi: partendo dalla suddivisione in livelli seguendo il pattern MVC e poi proseguendo nella definizione di ogni livello nello specifico.

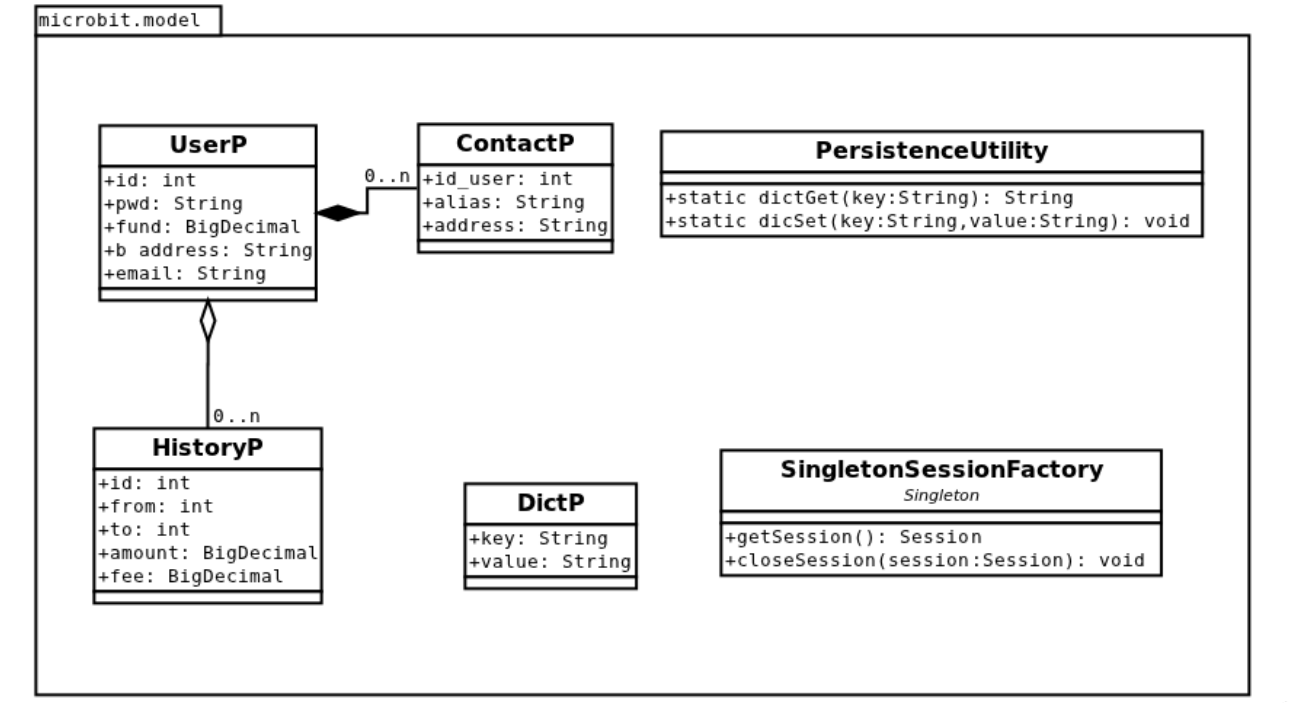


***Immagine 3**: class diagram di microbit.model.*

Dal diagramma ER relativo allo schema del database derivano direttamente le classi del package *microbit.model*; esse sono una mappatura 1:1 delle tabelle. Sono evidenziate le relazioni, anch'esse definite nel diagramma ER e nello schema del database. Possiamo notare come la relazione UserP – ContactP sia una composizione, mentre la relazione UserP – HistoryP sia una semplice aggregazione. Il motivo della differenza risiede nel fatto che la rubrica appartiene solo a un utente, e il resto del sistema non è affetto dalla sua esistenza o meno; invece, un oggetto HistoryP può coinvolgere due diversi utenti e mantiene informazioni che riguardano lo spostamento di valuta nel sistema e verso l'esterno.

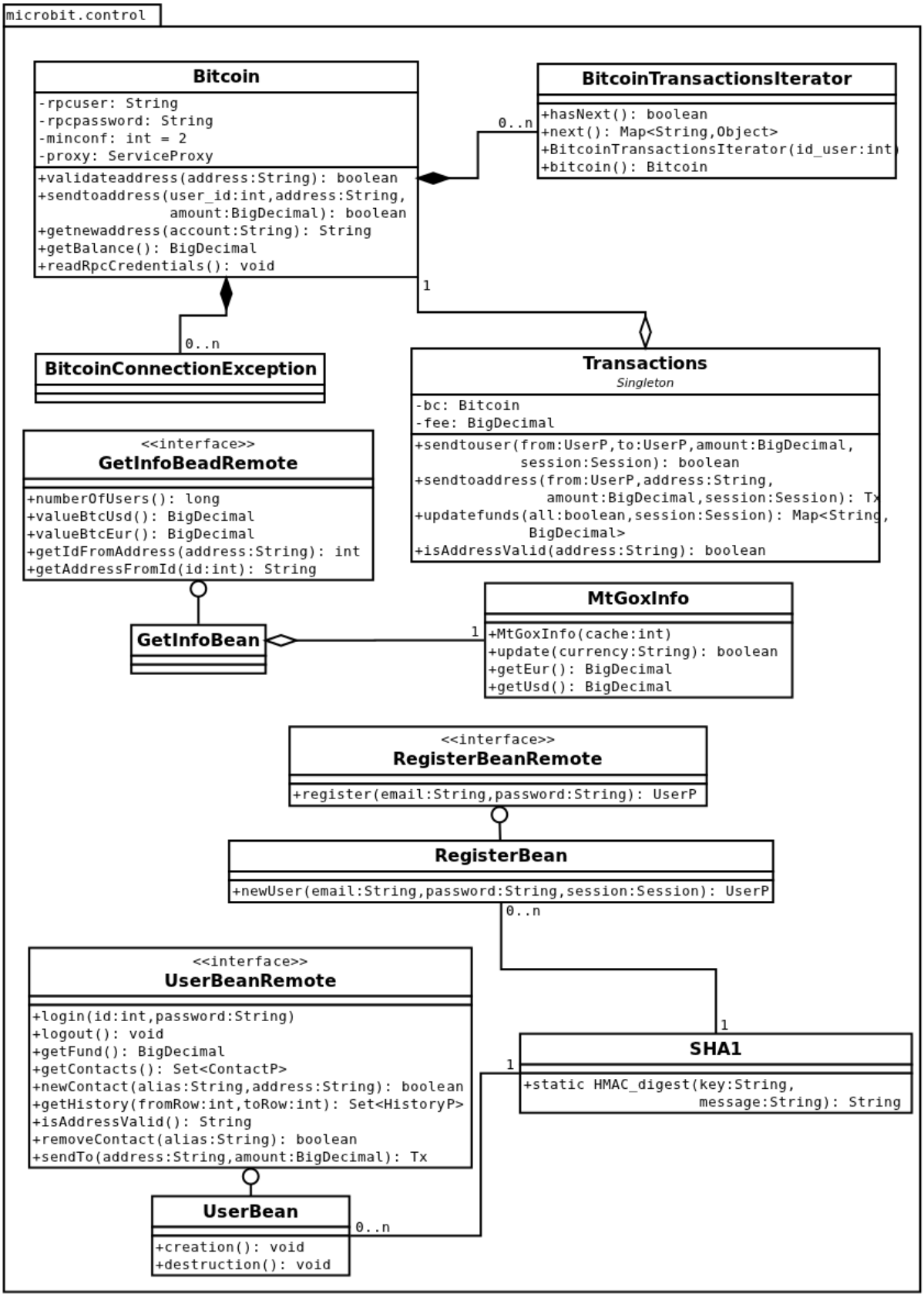


***Immagine 4**: class diagram del package microbit.control*

Il package microbit.control racchiude la business logic della applicazione. Esso è formato da alcune normali classi Java e da altre classi che implementano degli Enterprise Java Bean; queste ultime hanno il suffisso Bean nel nome. Fra le altre classi possiamo notare quelle che costituiscono l'interfaccia con il client Bitcoin (*Bitcoin*, *BitcoinTransactionIterator*, *BitcoinConnectionException* e *Transactions*); l'interfaccia con il sito MtGox.com e una che fornisce una primitiva per calcolare l'HMAC.

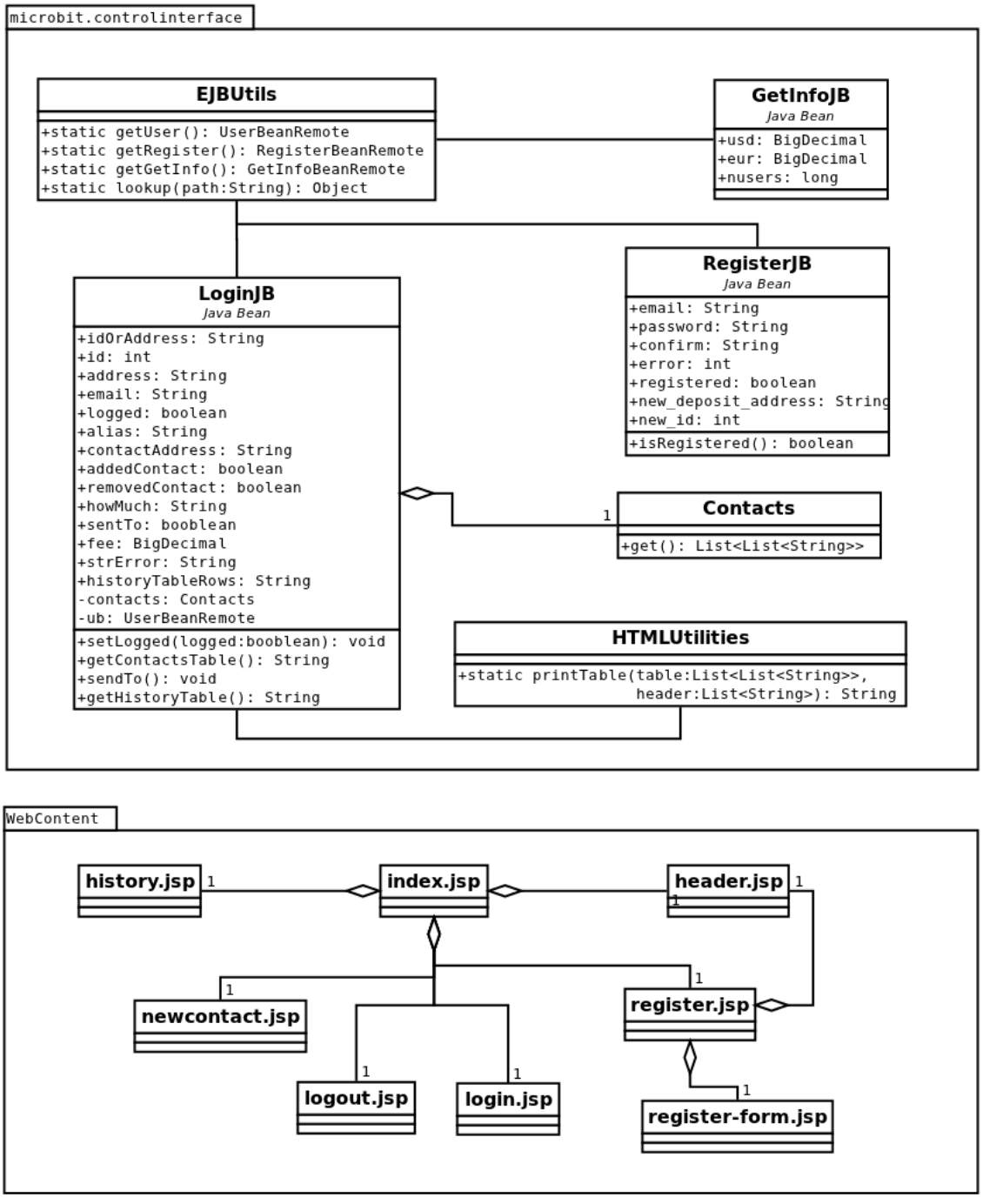


***Immagine 5**: class diagram del package microbit.controlinterface e delle pagine JSP*

L'interfaccia è suddivisa in pagine JSP e in classi Java (Java Bean e comuni classi) per ridurre al minimo il codice Java all'interno delle pagine JSP. Gli oggetti Java Bean sono parzialmente integrati con le pagine JSP e permettono un'integrazione abbastanza elegante se progettati in maniera opportuna.

Organizzazione del deployment dei vari livelli logici

Il livello persistente

Lo strato persistente è basato su Hibernate e MySQL. La mappatura fra classi POJO e tabelle dello schema del database è stata effettuata manualmente tramite dei file XML.

```
<hibernate-mapping package="org.silix.the9ull.microbit.model">
<class name="UserP" table="User">
<id name="id" type="int" column="`id`" >
<generator class="identity"/>
</id>

<property name="deposit_address" type="string" unique="true">
      <column name="`deposit_address`" length="40"/>
```

***Codice 1**: parte del codice XML che definisce la mappatura per la tabella User e gli oggetti UserP.*

Bisogna porre particolare attenzione all'uso delle sessioni Hibernate: usare lo stesso oggetto persistente in due diverse sessioni contemporaneamente può creare delle inconsistenze nella base di dati e gli aggiornamenti effettuati sugli attributi di un oggetto non saranno notificati all'altro.

Per questo motivo nel progetto viene usata una sessione globale per tutte le istanze dell'EJB che gestisce la sessione di utente connesso; infatti ogni utente può effettuare delle transazioni che modificano il campo *fund* e la tabella *history* di un altro utente del sistema. Per quanto riguarda i Session Bean Stateless, essi possono utilizzare delle sessioni indipendenti in quanto essi non modificano degli oggetti che vengono condivisi con altre entità.

Hibernate permette di descrivere gli attributi in modo accurato, permette così di garantire la consistenza dei dati nel database. Ad esempio, nel codice XML riportato si può osservare che è stato indicato il limite di dimensione del campo *deposit_address*.

Application server

Per il deployment è stato utilizzato l'*Application server* Glassfish. Esso si integra perfettamente nell'ambiente di sviluppo Eclipse Juno EE; lo stesso server Glassfish è stato installato automaticamente in fase di setup del progetto.

Per implementare il requisito non funzionale che riguarda la sicurezza è necessario abilitare la crittografia SSL nella configurazione dell'application server; in questo modo verrà utilizzato HTTPS che, oltre alla crittografia a chiave simmetrica per proteggere i contenuti scambiati con il server, permette anche di verificare l'identità del server mediante un certificato da acquistare da una delle CA commerciali riconosciute. Utilizzare un certificato SSL sul server permette di prevenire l'attacco *Man in the middle* [4]. Nell'ambiente di test è stato configurato l'accesso tramite il protocollo HTTPS; ma, data la mancanza dei certificati firmati da una CA, non è stato possibile testare il *setup* completo.

Interfaccia

Per quanto riguarda l'interfaccia, si è scelto di utilizzare la tecnologia JSP supportata da classi Java Bean.

La tecnologia JSP supporta in modo egregio l'inserimento di codice HTML/XHTML nelle pagine dinamiche, penalizzando però l'inserimento del codice. Esso, infatti, immerso nelle pagine HTML risulta illeggibile e non particolarmente mantenibile.

L'interfaccia del sito Microbit è stata quindi progettata per sfruttare i vantaggi della tecnologia e mitigare i difetti. Questo è stato possibile grazie all'integrazione di JSP con le classi Java Bean: è possibile inserire in modo elegante in una pagina JSP un attributo di una classe Java Bean (tramite il costrutto **<jsp:getProperty name=“ ” property=“ ” />**). Le classi Java Bean sono servite da *glue code* con la business logic del package *microbit.control*.

Pattern utilizzati

Model, View, Controller

Il pattern utilizzato per la struttura globale del progetto è il classico MVC, adatto soprattutto ad ambienti di programmazione ad oggetti. I package Java creati rispettano proprio questa divisione ed esso emerge in ogni operazione che si può effettuare. L'utente utilizza l'*interfaccia* (formata dal package *controlinterface* e dalle pagine JSP) per impartire un comando; il *controller* (essa non può, in nessun modo, interagire direttamente con il *modello*) acquisisce i comandi e li processa, scambiando, eventualmente, dati con il modello; infine il controller fornisce il risultato all'interfaccia, che lo visualizza.

Si può osservare che, anche all'interno dell'interfaccia, esiste una organizzazione del codice che segue il pattern MVC: le pagine JSP effettuano delle richieste alle classi Java Bean sottostanti che, dopo aver interagito con i livelli sottostanti (mediante l'interfaccia degli EJB), forniscono l'esito da mostrare all'utente.

Sotto-moduli del progetto implementano altri pattern.

DAO e DTO

Per quanto riguarda il livello di persistenza, i pattern sono implementati direttamente da Hibernate: esso fornisce un'astrazione del database, nascondendone i dettagli (peculiarità del pattern Data Access Object); in seconda istanza, esso utilizza delle classi POJO (Plain Old Java Object) per rappresentare i singoli oggetti persistenti del database. Questa ultima caratteristica implementa il pattern Data Transfer Object.

Session façade

Questo pattern viene implementato dalle specifiche JavaEE: la suddivisione del codice in business logic e interfaccia obbliga l'analista programmatore a progettare un'interfaccia che nasconda i dettagli implementativi della business logic ed esponga le funzionalità dei moduli.

Business delegate

Il pattern Business delegate è stato utilizzato nella progettazione dell'integrazione con il sito MtGox che prevede di fornire all'utente il valore aggiornato della valuta Bitcoin.

Effettuare il reperimento dei dati all'interno della *business logic* (e non, per esempio, direttamente

all'interno dell'interfaccia web, mediante del codice Javascript) ha avuto dei vantaggi:

- l'interfaccia con il client è più uniforme e più completa; l'implementazione del client risulta quindi più semplice;
- le problematiche di reperimento vengono centralizzate nel livello logico e la loro gestione risulta più semplice;
- la performance ne trae dei vantaggi, sollevando il client da operazioni superflue, come la connessione con ulteriori server e servizi esterni;
- la connessione ad ulteriori servizi esterni potrebbe entrare in conflitto con le policy di sicurezza della rete;
- la centralizzazione ha reso possibile l'implementazione del caching; questo influisce positivamente sulla performance e sul minor utilizzo di risorse. Nel caso specifico, i valori dei prezzi dei Bitcoin vengono aggiornati dal sito MtGox ogni 10 minuti, intervallo di tempo in cui non si verificano variazioni significative.

Singleton

Questo pattern è nato con la programmazione ad oggetti, esso permette di avere un oggetto globale, accessibile da ogni istanza degli oggetti dell'applicazione.

È stato applicato alle classi *SingletonSectionFactory* e *Transactions*.

La prima classe gestisce le sessioni Hibernate dello strato persistente: ogni altro oggetto deve poter ottenere una sessione per effettuare operazioni sugli oggetti persistenti, e più di un oggetto FactorySession avrebbe rappresentato un inutile spreco di risorse.

La classe *Transactions* si interfaccia con il client Bitcoin. Considerando che esiste un unico client e che esso richiede una configurazione specifica per poter essere usato (reperendo le credenziali di accesso dal file di configurazione dello stesso), il pattern Singleton si è rilevato adatto.

Progettazione dell'interfaccia

Per quanto riguarda l'interfaccia, le linee guida seguite nell'implementazione prevedono delle pagine semplici e veloci; pagine minimali che permettano di impartire i comandi e di visualizzare l'esito in modo chiaro e immediato.

Quando possibile, si sono evitati anche i bottoni, obbligando l'utente ad utilizzare il tasto Enter per confermare i dati inseriti in alcune form.

Le comunicazioni all'utente sono dei semplici messaggi in font nero su sfondo bianco; l'unica eccezione riguarda i messaggi di errore o non ordinari, che sono invece in rosso, per trasmettere all'utente un senso di urgenza ed attirare l'attenzione.

Un'eccezione al minimalismo dell'interfaccia coinvolge un simpatico logo animato all'inizio della pagina.

L'interfaccia che si presenta caricando la pagina principale è riportata nell'immagine 6:

***Immagine 6**: interfaccio della home page. È possibile raggiungere la pagine per creare un nuovo utente, oppure inserire i dati per autenticarsi. In tutte le pagine vengono riportate le informazioni riguardo il valore della valuta Bitcoin e il numero di utenti del sito Microbit.*

La pagina register (register.jsp) dispone di una semplice form in cui inserire il proprio indirizzo email e la password. Quando la registrazione va a buon fine vengono assegnati un user id e un indirizzo Bitcoin per poter accreditare valuta.

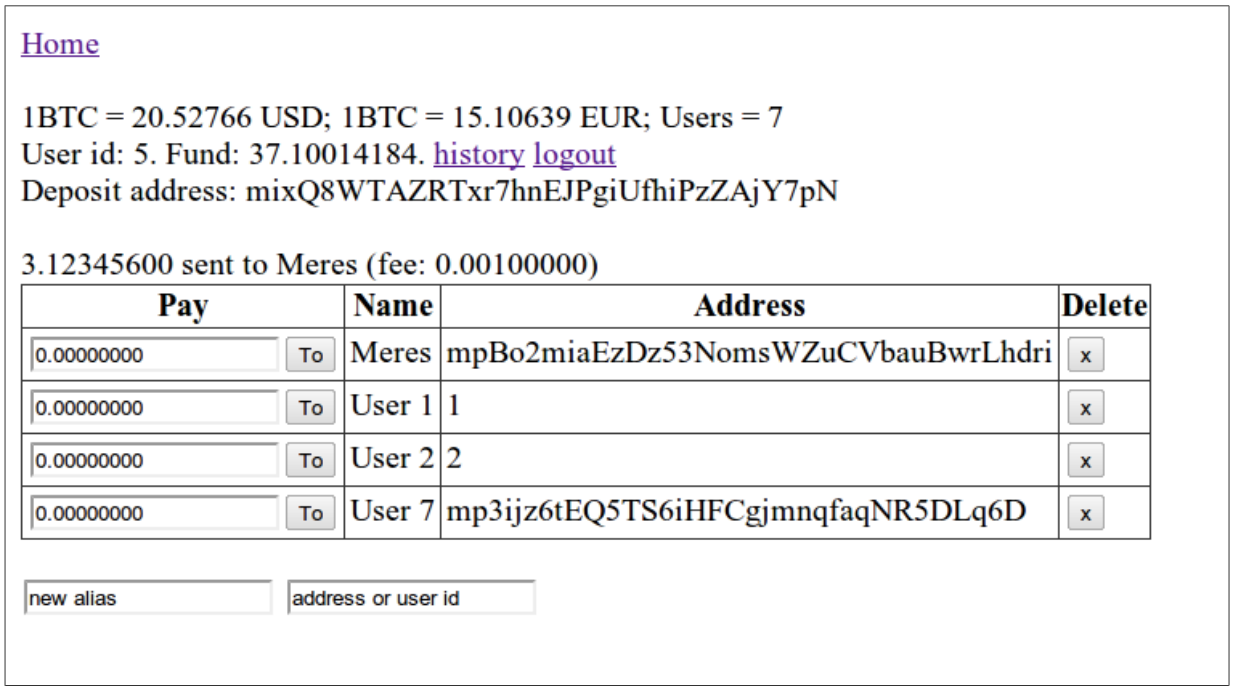
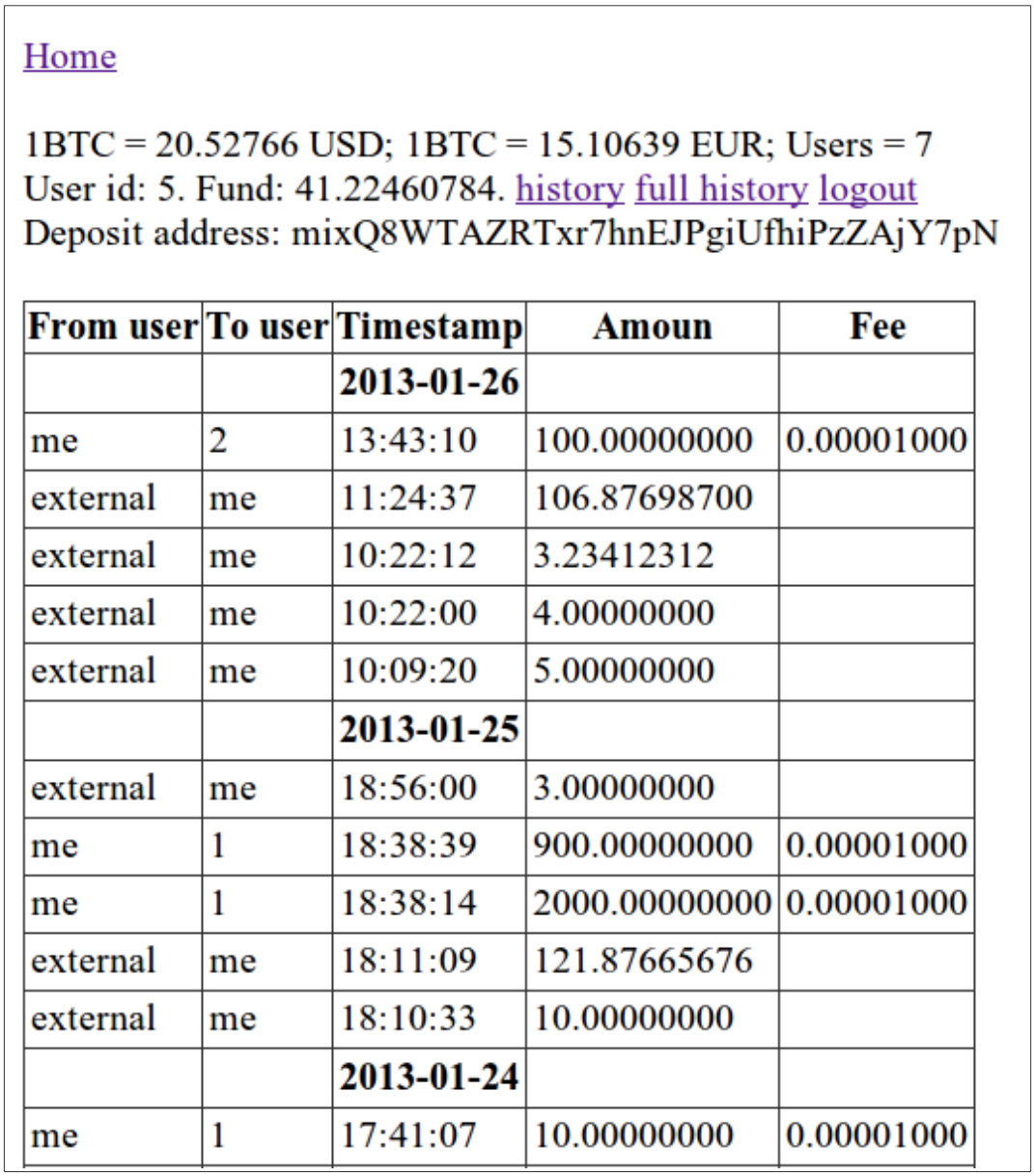


***Immagine 7**: interfaccia della rubrica, mostrata subito dopo il login. Prima della tabella è visibile il messaggio di conferma di avvenuto pagamento di 3,123456 Bitcoin al contatto Meres. Si può notare che nella colonna Address si possono inserire sia indirizzi Bitcoin, sia user id del sistema Microbit.*



Home

1BTC = 20.52766 USD; 1BTC = 15.10639 EUR; Users = 7
User id: 5. Fund: 41.22460784. history full history logout
Deposit address: mixQ8WTAZRTxr7hnEJPgiUfhiPzZAjY7pN

From user	To user	Timestamp	Amoun	Fee
		2013-01-26		
me	2	13:43:10	100.00000000	0.00001000
external	me	11:24:37	106.87698700	
external	me	10:22:12	3.23412312	
external	me	10:22:00	4.00000000	
external	me	10:09:20	5.00000000	
		2013-01-25		
external	me	18:56:00	3.00000000	
me	1	18:38:39	900.00000000	0.00001000
me	1	18:38:14	2000.00000000	0.00001000
external	me	18:11:09	121.87665676	
external	me	18:10:33	10.00000000	
		2013-01-24		
me	1	17:41:07	10.00000000	0.00001000

***Immagine 8**: visualizzazione della history. Gli indirizzi Bitcoin esterni non vengono memorizzati, viene solo indicata la natura del trasferimento. Diversamente, negli altri casi è possibile visualizzare l'user id.*

Implementazione e deployment

Gestione dello stato dell'applicazione

L'applicazione deve gestire lo stato essenzialmente a due diversi livelli: a livello di core business e a livello di Servlet web (e quindi pagine JSP).

Le classi Stateful Session Bean

Nel primo caso si utilizzano i Session Bean: ne esistono infatti di due tipi in base alla gestione o

meno dello stato dell'utente specifico.

Utilizzare delle Session Bean Stateless è meno dispendioso dal punto di vista delle risorse, quindi è opportuno scegliere al meglio l'organizzazione del codice in Session Bean.

Per quanto riguarda il progetto Microbit è stata utilizzata una sola classe Session Bean Stateful: quella che mantiene lo stato dell'utente. Essa gestisce tutte le interazioni, dal **login** al **logout**; inoltre esso supporta la serializzazione, che permette all'*application server* di gestire al meglio la memoria fisica: esso immagazzinerà su disco i Bean Stateful in caso di necessità mediante una procedura di swap.

Per supportare la serializzazione è necessario che gli attributi serializzabili della classe implementino l'interfaccia *Serializable* (ad esclusione degli attributi di tipo nativo, ovviamente), e che dei metodi con le annotazioni opportune si occupino della gestione degli altri attributi (nel nostro caso, la sessione Hibernate).

Sessioni web

Il secondo livello in cui è possibile gestire lo stato è a livello web, mediante i cookie che si possono memorizzare nel browser dell'utente.

L'architettura EE utilizza il cookie SESSION in modo automatico per dividere le richieste per utente. Le Servlet (e quindi pagine JSP) sono legate alla sessione del browser e mantengono lo stato finché la sessione del browser è valida, compatibilmente con lo scope assegnato al Java Bean.

Gli scope utilizzati nel progetto sono:

- application: per i Java Bean condivisibili fra tutti gli utenti che utilizzano l'applicazione contemporaneamente;
- session: per i Java Bean che gestiscono i dati della sessione utente, dal login al logout.

Nel progetto non utilizza altri cookie per memorizzare dati, si fa affidamento solo allo strato fornito dalle Servlet anche per memorizzare i dati relativi all'interfaccia.

La sessione web e i Session Bean Stateful vengono sincronizzati dal framework Java EE.

Transazioni

Le uniche transazioni utilizzate dal progetto riguardano l'accesso al database.

Livello di persistenza

Il framework Hibernate mette a disposizione gli oggetti *Transaction* per implementarle. Dopo aver iniziato una transazione ed effettuato le modifiche agli oggetti persistenti, esse possono terminare in 3 modi diversi:

- *transaction.commit()*: le modifiche apportate vengono confermate e rese persistenti;
- *transaction.rollback()*: le modifiche vengono annullate e il database riportato allo stato consistente precedente alla transazione;
- interruzione dell'applicazione: per un evento esterno o per un bug il programma termina senza terminare la transazione. Il database verrà riportato automaticamente allo stato precedente alla transazione.

Ogni transazione ha una durata temporale molto limitata, essa non supera mai l'unica richiesta

HTTP.

Livello logico

A livello logico non vengono usate transazioni perché non sono stati implementati meccanismi che richiedono di mantenere lo stato più a lungo di un'unica richiesta HTTP (e quindi chiamata al livello logico EJB). Nel caso in cui questo fosse richiesto bisogna considerare che le transazioni semplificano molto il controlli riguardanti le possibili azioni intraprese dall'utente.

Tecnologie usate

JSON-RPC

Oltre alle tecnologie direttamente legate all'architettura, è stata utilizzata una libreria che implementa un client json-rpc chiamata jj1 [5]. Essa si occupa di fornire degli oggetti Java dinamici partendo dagli oggetti Json ottenuti dalle chiamate RPC.

Essa è stata utilizzata per implementare due diverse funzionalità:

- per interfacciarsi con il client Bitcoin;
- per la comunicazione con il sito MtGox.com.

Json è molto utilizzato per lo scambio di dati; le sue caratteristiche principali sono semplicità e indipendenza dal linguaggio di programmazione.

HMAC

Significa Keyed-Hash Message Authentication Code [6] ed stato progettato per l'autenticare e verificare l'integrità di un messaggio. All'interno del progetto, HMAC ha sostituito la funzione di hash per memorizzare le password degli utenti all'interno del database.

L'algoritmo HMAC viene richiamato utilizzando come messaggio la password dell'utente in chiaro, e come chiave l'id dell'utente a cui appartiene la password.

L'hash calcolato in questo modo ha la peculiarità di essere legato all'utente a cui si riferisce; questo fatto ha due conseguenze: utenti che utilizzano la stessa password non sono riconoscibili osservando l'hash codificato; e, fatto più rilevante, gli attacchi di cracking mediante *rainbow table* [7] non sono possibili.

Un attacco rainbow table consiste nell'utilizzare una tabella hash pre-computata che associa ad ogni possibile hash la password che lo ha generato. Legando l'hash al uno specifico id utente, è necessario generare una diversa tabella per ogni utente, questo rende l'attacco impraticabile.

GIT

Git è un software di controllo versione distribuito complesso e potente. Il repository è ospitato su github.com [8] e il codice disponibile alla comunità sulla piattaforma di social coding. Ogni modifica del software è stata opportunamente commentata usando gli strumenti messi a disposizione da git.

Deployment

Uno dei vantaggi dell'architettura EE è la possibilità di scalare l'applicazione su più server. È

possibile utilizzare fino a 3 server senza riprogettare l'applicazione:

- il primo per il DBMS MySQL: viene usato tramite un socket di rete, è sufficiente modificare il file XML di configurazione di Hibernate inserendo l'indirizzo del server MySQL;
- il secondo per il livello logico EJB. Esso espone delle interfacce remote identificate attraverso dei path JNDI (Java Naming and Directory Interface);
- il terzo per l'interfaccia web e le Servlet.

Il server fisico che ospita le Servlet è l'unico che necessita l'accesso alla rete. Per una corretta gestione della sicurezza è opportuno che l'altro, o gli altri, server non siano accessibili dalla rete in modo diretto. È possibile ottenere questo effetto mediante delle opportune regole del firewall, oppure realizzando una rete privata fra i server che gestiscono l'applicazione. Una scelta di questo tipo prevede la progettazione di una eventuale metodologia per gestire la manutenzione remota.

Deployment su localhost

Per quanto riguarda i test in *localhost*, è stato installato un server Glassfish per testare l'applicazione durante lo sviluppo. Esso è stato configurato e viene gestito da Eclipse. Ad ogni modifica del codice, Eclipse si occupa di ripubblicare l'applicazione aggiornata sul server locale.

Giunti alla versione finale dell'applicazione web, si è provveduto all'installazione del server Glassfish direttamente nel sistema operativo. Dopo aver avviato manualmente il server, è stato installato il pacchetto **.war** contenente l'applicazione mediante l'interfaccia web di amministrazione.

Comando per avviare il server Glassfish:

```
sudo ~/opt/glassfish3/glassfish/bin/asadmin start-domain
```

Comando per terminare il server:

```
sudo ~/opt/glassfish3/glassfish/bin/asadmin stop-domain
```

Problematiche affrontate nello sviluppo

Durante lo sviluppo è stato utilizzato un blog per tenere traccia delle problematiche incontrate e delle relative soluzioni [9].

- La parte più complessa dello sviluppo ha riguardato la comunicazione fra il livello business logic e il livello interfaccia. Inizialmente si era identificato TomEE come application server da utilizzare, scelto per la pubblicizzata leggerezza e semplicità. Successivamente, l'impossibilità di risolvere le problematiche riscontrate, si è scelto di migrare verso il prodotto di Oracle Glassfish, più supportato dalla comunità. Le risorse online che riguardano TomEE sono piuttosto carenti rispetto a Jboss e Glassfish.
- Un secondo problema emerso riguarda il debugging delle codice contenuto nelle pagine JSP: spesso si ottengono degli errori generici che fanno riferimento alla presunta invalidità della classe, senza indicazioni più precise. Il metodo più veloce per risolvere il problema si è dimostrato riportare il codice in una classe Java; effettuare il debugging sulla stessa porterà quasi sicuramente a una veloce individuazione del problema.
- Un altro problema è emerso con le transazioni in Bitcoin: inizialmente è stata usata la primitiva del client Bitcoin **sendtoaddress**; essa ritorna un identificativo della transazione. Per ottenere le altre informazioni utili (come, ad esempio, il costo applicato alla transazione), era necessario ottenere dal client Bitcoin una lista delle transazioni eseguite e ricercare quella appena effettuata. Questa tecnica potrebbe risultare terribilmente lenta al

crescere delle transazioni effettuate dal client. Controllare solo le transazioni recenti può comunque diventare un problema al crescere degli utenti del sito. La soluzione è stata trovata in una seconda primitiva messa a disposizione da Bitcoin: **sendfrom**. Essa lega la transazione ad un account indicato e quindi permette di filtrare le transazioni ed effettuare la ricerca solo fra quelle effettuate dall'utente. Inoltre, è stato realizzato uno speciale iteratore (*BitcoinTransactionsIterator*) che permette di caricare un limitato numero di transazioni invece dell'intera lista, che nel tempo potrebbe diventare ingombrante e rallentare l'operazione.

Bibliografia

[1] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto

[2] http://mtgox.com

[3] https://en.bitcoin.it/wiki/Transaction_fee#Rules_for_calculating_minimum_fees

[4] http://it.wikipedia.org/wiki/Attacco_man_in_the_middle

[5] http://code.google.com/p/jj1/

[6] http://it.wikipedia.org/wiki/HMAC

[7] http://it.wikipedia.org/wiki/Tabella_arcobaleno

[8] https://github.com/the9ull/microbit

[9] http://the9ull.silix.org/wiki/Web_architectures_project_blog