23/24

Progetto di Web Systems design and architecture

PROFESSORE LA CASCIA MARCO

A CURA DI FLAVIO GIUSEPPE AMATO - MATRICOLA N.RO 0767067

Indice

Consegna progetto

Si richiede la realizzazione di una web application in grado di gestire dei conti associati a delle carte e di effettuare operazioni base quali accredito e addebito.

Entrando più in dettaglio, il sistema distinguerà tre tipi di utenti (CLASSIC_USER, MERCHANT, ADMIN) ognuno dei quali può accedere a funzionalità specifiche del ruolo:

- Il classic user può: verificare lo stato del proprio conto e visualizzare gli ultimi 5 movimenti associati alla sua carta e generare un report secondo alcuni criteri specifici.
- Il merchant può: effettuare operazioni di accredito e addebito noto il numero di carta e generare un report con le operazioni che ha eseguito secondo alcuni criteri specifici.
- L'admin può: creare una nuova carta, bloccare e sbloccare una carta noto il numero, inserire nuovi utenti di tutte le tipologie, disabilitare i negozianti, generare un report delle operazioni eseguite nel sistema secondo alcuni criteri specifici.

Tutti gli utenti possono, ovviamente, effettuare il login e il logout.

Requisiti del progetto

Il progetto è stato realizzato utilizzando le seguenti tecnologie:

- JDK 21.0.1;
- Tomcat 10.1.15;
- Database: MySql;
- Maven per gestire le dipendenze e importarle automaticamente senza includere manualmente i jar, nello specifico maven-compiler 3.8.0.

Run Progetto: How to

Per eseguire il codice da un IDE bisogna:

1. Importare il progetto nel proprio IDE.
2. Buildare il progetto con il comando maven build.
3. Creare il db nel proprio DBMS, nel mio caso MySQL, col nome "amato_wsda". ***Nota**: è possibile scegliere un nome alternativo per il db, ma è necessario modificare il persistence.xml andando a modificare tutte le occorrenze di "amato_wsda" col nome scelto.*
4. Impostare il value della `<property name="hibernate.hbm2ddl.auto" value="" />` a "create" nel persistence.xml (così hibernate creerà in automatico le tabelle).
5. Scaricare TomcatServer 10.1.15.
6. Integrare il server appena scaricato nel proprio IDE (nel mio caso IntelliJ IDEA).
7. Collegare il progetto al server.
8. Eseguire il progetto sul server una prima volta, successivamente verificare la corretta creazione delle tabelle.
9. Arrestare il server.
10. Eseguire initDB.sql sul DBMS per aggiungere utenti e carte al sistema.
11. Se tutti questi passi sono andati a buon fine, sarà possibile accedere con le credenziali degli utenti specificate nel file .sql.

Codice

1. Struttura progetto

In questa sezione descriveremo la struttura del progetto evidenziando le directory più rilevanti.
***Nota**: Il contenuto delle directory ed i file contrassegnati da un asterisco verranno descritto nel dettaglio in seguito*

Root:

```
finalproject_lastversion/
├── src/
        ├──main/
            ├── java/
                ├── it/
                    ├── finalproject_lastversion/
                        ├──DAO/*
                        ├──DTO/*
                        ├──model/*
                        ├──service/*
                        ├──utils/*
                        ├──web/
                            ├──filter/*
                            ├──servlet/*
            ├── resources/
                ├──META-INF/
                    ├── persistence.xml*
            ├── webapp/*
        ├── pom.xml*
        ├── initDB.sql*
```

1.1 pom.xml

Il pom contiene tutte le dipendenze del progetto che verranno automaticamente scaricate eseguendo il comando build di maven sul progetto.

Le dipendenze dichiarate nel pom sono le seguenti:

- **jakarta ee web api** (10.0): la versione di jakarta enterprise edition utilizzata
- **apache pdfbox**: libreria di utilità per la generazione dei pdf (utile per generare i report dalla servlet)
- **jakarta servlet jsp jstl**: le jakarta standard tag library, utili nelle jsp per rendere il codice più leggibile ed evitare l'utilizzo di scriptlet (infatti importando le tag library è

possibile utilizzare comandi condizionali con una sintassi molto più compatta e human readable)
- **org glassfish web**: dipendenza di utilità che consente alle jstl di funzionare in modo corretto
- **commons lang di apache**: utile per la validazione dei campi lato servlet (soprattutto coi metodi isBlankAny dello StringUtils e isCreatable di NumberUtils che verifica se una stringa è parsabile come numero)
- **jakarta servlet** (6.0.0)
- **jakarta jsp** (3.1.0)
- **jakarta el** (4.0.0): consente di valutare espressioni all'interno di jsp, permettendo l'accesso e la manipolazione dei dati in modo più semplice ed efficace
- **mySql connector** (8.1.0): consente la connessione con il DB
- **jakarta validation** (3.0.2): consente di inserire delle annotation contrassegnate dalla @ per effettuare la validazione dei modelli per l'inserimento nel DB (es. @NotNull)
- **jakarta annotation**(2.1.1): consente di specificare delle annotation all'interno delle classi per consentirne la loro categorizzazione (@WebServlet, @WebFilter …)
- **jakarta persistence**(3.1.1): utilizzata principalmente per la mappatura oggetto-relazionale (ORM), consentendo agli sviluppatori di gestire dati persistenti in un database relazionale utilizzando oggetti Java. ***N.B.:*** *non cambiare la versione, jakarta persistence 3.1.1 ha aggiunto la possibilità di generare l'id con GenerationType.UUID*
- **hibernate**(6.1.7): persistence provider di jakarta persistence, si occupa di interrogare il database dietro le quinte e restituirne i risultati.

1.2 Creazione_DB_lastversion.sql e initDB

Creazione_DB è responsabile della creazione del DB. InitDB popola le tabelle inserendo: 10 utenti, 6 carte e 30 transazioni (di cui due in stato creato) per testare tutte le funzionalità presenti all'interno dell'applicativo.

Nello specifico, initDB aggiungerà:

- n. 1 admin con credenziali **username:** admin, **password:** admin123
- n. 3 merchant con credenziali **username:** merchantx, **password:** merchantx con x= 1,2,3
- n. 6 user classici con credenziali **username:** utentey, **password:** utente0y con y = 1…6
- n. 6 carte associate agli user classici con numero 111..1, 222..2, 333..3 e così via con 16 occorrenze dello stesso numero.
- n.30 transazioni associate in modo casuale ad una carta e con data casuale negli ultimi due anni. Sono state inserite due transazioni in stato = "STATE_CREATED" a carico dell'utente merchant1 per dimostrare il corretto funzionamento dell'avviso che il sistema mostra al merchant, se quest'ultimo possiede delle transazioni in attesa di conferma.

1.3 persistence.xml

Il persistence.xml consente all'ORM provider di capire quali siano i model da mappare nel DB (tag <class>) e specifica properties che consentono ad hibernate di conoscere l'URL del db, il cui valore è la stringa "jdbc:mysql://localhost3306/nome_schema"; nome_schema sarà lo schema creato nel DBMS. Inoltre sono necessari i campi username e password (nel mio caso root root) per accedere al DB.

Hibernate o comunque l'ORM provider fa uso di un entityManager per gestire le interazioni col DB.

```
<persistence-unit name="wsda-unit"
    transaction-type="RESOURCE_LOCAL">
```

Questo tag stabilisce quale sia il nome dell'unità di persistenza e sarà utilizzato da jakarta.Persistence per creare un entityManagerFactory; l'entityManagerFactory farà injection della connessione quando le classi del package service interagiranno col database. Di fatti, ogni metodo implementato del service ha le seguenti righe di codice:

```
EntityManager entityManager =
LocalEntityManagerFactoryListener.getEntityManager();
```

Tra le property importanti, troviamo:

```
1. <property name="hibernate.show_sql" value="true" />
2. <property name="hibernate.hbm2ddl.auto" value="" />
```

Il **primo** mostra nella console del server locale le query jpql eseguite durante il run del progetto.

Il **secondo** può assumere diversi valori tra cui update, create, create-drop e specifica quello che hibernate debba fare sul db *(aggiornare la struttura delle tabelle, crearle, cancellarle se esistono e crearle e così via..)*. Nel caso in cui si esegua il progetto per la prima volta, impostare il valore a create e poi modificarlo.

1.4 webapp

Il contenuto della cartella webapp è suddivisso in diverse sottocartelle che contengono principalmente le view:

- admin, merchant, user contengono le jsp accessibili da utenti admin, utenti merchant ed utenti classici. È presente anche una cartella public contenente jsp accessibili da chiunque (login.jsp, index.jsp)
- js contiene uno script javascript con due funzioni definite ed usate in diverse pagine
- assets contiene file di utilità di bootstrap ed una cartella images contenente il logo dell'Università degli Studi di Palermo

Vi è anche un file libero navbar.jsp che è un common import in tutte le pagine.

1.5 model

Il package model contiene: *(le annotation presenti nelle definizioni della classe esprimono il nome della tabella, i nomi delle colonne, i vincoli di chiave esterna)*

- **User**: classe che sarà mappata nella tabella user del db. Non è stata inserita un riferimento alla carta per evitare di avere un campo che per due tipi di utenti su tre sarà null.
- **Transaction**: classe che sarà mappata nella tabella transaction del db, contiene un riferimento allo user che ha eseguito la transazione e alla carta su cui l'operazione è stata eseguita.
- **CreditCard**: classe che sarà mappata nella tabella credit_card del db, possiede un riferimento al proprietario che sarà uno user e un set di transazioni contenente le operazioni eseguite su quella carta.

Nei model sono presenti anche due enum: uno **UserRole** che specifica il ruolo dello user, e un **TransactionState** che specifica lo stato della transazione.

1.6 DTO

Il pattern DTO (Data Transfer Object) è un design pattern utilizzato nell'ambito dello sviluppo software per gestire la trasmissione di dati tra componenti di un'applicazione. Il suo obiettivo principale è quello di semplificare la comunicazione e ridurre la complessità associata alle interazioni tra componenti. Un DTO, quindi, è una classe con design ad hoc per le necessità dell'applicativo. Anche se i suoi obiettivi sono utili per applicazioni che hanno una portata sicuramente più grande di questo progetto, lo UserDTO rappresenta un buon esempio; di fatti, questi non contiene la password e rappresenta l'utente messo in sessione al login.

Il package DTO contiene: **UserDTO**, **CreditCardDTO**, **TransactionDTO** che sono repliche delle classi dei model (prive di annotation e, talvolta, di alcuni campi e rappresentano le istanze delle classi inviate in pagina).

In ognuna di queste classi sarà presente un metodo, la cui firma è:

```
public static classDTO createClassDTOFromModel(Class
instance);
```

Questo consente di creare un DTO a partire da un model (che potrebbe essere il risultato di un'interrogazione al database).

1.7 utils

Il package utils contiene tre classi java di utilità:

1. **LocalEntityManagerFactoryListener**: annotata con @WebListener, implementa l'interfaccia ServletContextListener che definisce due metodi specifici cioè contextInitialized e contextDestroyed, lanciati rispettivamente al run della web app e alla sua chiusura.

   ```
   @Override
   public void contextInitialized(ServletContextEvent sce) {
           try {
               entityManagerFactory =
   Persistence.createEntityManagerFactory("wsda-unit");
           } catch (Throwable ex) {
               System.err.println("Initial SessionFactory
   creation failed." + ex);
               throw new ExceptionInInitializerError(ex);
           }
       }
   ```

   Il metodo sopra intercetta l'avvio dell'app e esegue il corpo del metodo istanziando una factory che ne garantisce l'unicità all'interno dell'applicativo.

   ```
   @Override
   public void contextDestroyed(ServletContextEvent sce) {
       if (entityManagerFactory != null) {
           entityManagerFactory.close();
   ```

```
        }
    }
```

Il contextDestroyed alla chiusura della web app distrugge la factory. Sono presenti altri metodi: uno ritorna l'entityManager che rappresenta una vera e propria connection e l'altro chiude la connection.

2. **UtilityConversion**: definita per alleggerire il codice della servlet. Presenta due metodi statici utili nell'istanziazione di una transaction. (utilizzata per la ricerca di transazioni, trova riscontro nel path shared/ExecuteReportServlet)
3. **ReportGeneratorUtil**: utilizzata nella servlet che genera il report, sfrutta la dipendenza apache pdfbox.

1.8 DAO

Il pattern DAO (Data Access Object) è un design pattern utilizzato per separare la logica di accesso ai dati (ad esempio, un database) dalla logica aziendale all'interno di un'applicazione. Il suo obiettivo principale è quello di fornire un'astrazione tra il codice che interagisce con i dati e il sottostante sistema di archiviazione dei dati, migliorando la manutenibilità e la flessibilità del codice.

La composizione del package DAO è la seguente:

- **IBaseDAO<T>:** un'interfaccia con un generic T che possiede la firma dei metodi CRUD base e un metodo setEntityManager utile per fare l'injection della connesione al DB
- **Sottopackage (card, transaction, user):** hanno pressoché la stessa struttura, ossia un'interfaccia che estende IBaseDAO e sostituisce al generic T il nome della classe (es. `CreditCardDAO extends IBaseDAO<CreditCard>`). Conterrà, inoltre, firme di metodi specifici che dovranno essere implementati dalla classe contenuta nello stesso sottopackage (es `CreditCardDAOImpl implements CardDAO`). Le classi che implementano l'interfaccia possiedono, inoltre, una variabile private EntityManager.

1.9 service

Il package service si compone di:

- **MyServiceFactory**: che contiene 3 metodi statici che restituiscono i rispettivi service, evitando che siano generati più istanze dello stesso service all'interno della stessa sessione *(concetto di singleton)*
- Tre sottopackage che possiedono:
    - Un'interfaccia che dichiara tutti i metodi del service specifico che prende il nome dalla entity con cui interagisce (es. CreditCardService)
    - Una classe che implementa il proprio service e possiede un campo private nonché un'istanza del corrispondente DAO, che si occuperà a tutti gli effetti di interrogare il DB (es. CreditCardServiceImpl implements CreditCardService)

1.10 Web

Il package web contiene due sottopackage:

1. **Filter**: contiene una sola classe CheckAuthFilter che si occupa di intercettare le richieste ed applicare le regole stabilite all'interno della stessa. E' una forma di

sicurezza che verifica che, il ruolo del richiedente di una risorsa, sia consono alla risorsa richiesta.

2. **Servlet**: si struttura in diversi sottopackage ed ha due classi "libere" che sono mappate nel path /public/. L'organizzazione dei path definita nel package servlet consente di realizzare un filtro in maniera abbastanza intuitiva. Di fatti se analizziamo il contenuto dei sottopackage avremo:
   - **auth**: si occupa di login e logout *(mappata nel path /public-il path è specificato accanto alla annotation @WebServlet)*
   - **admin**: si occupa di tutte le funzionalità relative all'admin *(path: /admin/…)*
   - **user**: si occupa di tutte le funzionalità relative allo user *(path: /user/…)*
   - **merchant**: si occupa di tutte le funzionalità relative al merchant *(path: /merchant/…)*
   - **share**: si occupa di tutte le funzionalità condivise *(path: /shared/…)*

Funzionalità

1. Public – funzionalità accessibili da chiunque

All'avvio dell'applicazione l'utente ha di fronte una schermata che presenta una barra di ricerca che consente di verificare il saldo a partire dal numero di carta; inoltre, è presente un link tramite il quale recarsi alla schermata di login (se non si è già autenticati).

La pagina di login consente di effettuare l'accesso tramite username e password e, se le credenziali sono corrette, il sistema indirizza l'utente alla propria are riservata. (scelta in base al ruolo)

2. User – funzionalità accessibili da classic user

La tua area riservata presenta due card che ti consentono di accedere a due funzionalità: la generazione di un report e la visualizzazione stato conto.

Nel caso in cui si scelga di generare un report, l'utente è reindirizzato ad una pagina con un form di ricerca che consente il filtraggio delle transazioni associate alla carta; il report sarà scaricato automaticamente al submit del form.

Se si vuole visualizzare lo stato conto, la pagina a cui si è rimandati possiede: una card(avente numero di carta e saldo) e una tabella con gli ultimi 5 movimenti associati alla carta.

3. Merchant – funzionalità accessibili da merchant

La tua area riservata presenta un form di ricerca da cui è possibile generare un report valorizzando i campi presenti nel form (filtrando di conseguenza le transazioni da te eseguite) e, al di sotto del form, è presente un link che che ti consente di accedere alla pagina da cui eseguire una transazione.

La pagina per eseguire una transazione presenta dei campi da valorizzare(nello specifico tipologia di operazione, importo e numero di carta). Se procediamo dopo aver valorizzato correttamente tutti i campi, il sistema mostra un riepilogo della transazione che permette di confermarla o negarla.

Funzionalità aggiuntiva: oltre al riepilogo della transazione, il sistema prevede la possibilità di gestire le transazioni rimaste in sospeso. Questa funzionalità è accessibile tramite la barra di navigazione del merchant e mostrerà una lista di transazioni in forma tabellare e due tasti conferma e nega. Al login, se il merchant ha operazioni in sospeso, un banner lo avviserà che ci sono transazioni in attesa di conferma.

4. Admin – funzionalità accessibili da admin

La tua area riservatà è composta da tre card che ti portano a tre diverse funzionalità: inserimento di una carta, ricerca di una carta, visualizzazione lista di tutti gli utenti merchant.

Dalla barra di navigazione è anche possibile: inserire un utente, generare un report delle transazioni con opzioni aggiuntive rispetto a quelle di user e merchant.

Analizziamo più specificatamente il flusso applicativo delle diverse funzionalità:

1. Inserimento di una carta: sarà necessario inserire il nome dell'utente a cui aggiungere la carta e confermare. Il sistema prevede che ogni user abbia al più una carta associata: l'aggiunzione della carta andrà a buon fine se l'utente è abilitato e se non possiede carta, altrimenti rimanda alla home con un messaggio di errore.
2. Ricerca di una carta: la pagina avrà un campo numero di carta da valorizzare. Inserendo un numero di carta presente nel DB, l'admin avrà di fronte il riepilogo della carta contenente: saldo, numero di carta e intestatario; dal riepilogo, si potrà abilitare o disabilitare la carta tramite un bottone, il cui valore dipenderà dallo stato della carta.
3. Visualizzazione lista merchant: il sistema mostra una pagina avente una lista di utenti merchant in forma tabellare, da cui li si può abilitare o disabilitare.
4. Inserimento utente: l'admin ha di fronte un form da valorizzare con i campi username, password e ruolo. Se la validazione del form va a buon fine e l'admin conferma, l'utente verrà aggiunto alla tabella user e potrà entrare nella web app. In caso di errore, l'admin verrà reindirizzato alla stessa pagina con un messaggio di errore.
5. Generazione report: il flusso è analogo a quello di user e merchant. L'unica differenza risiede nella possibilità di valorizzare due campi aggiuntivi presenti nel form, ossia numero di carta e username merchant.