

Keycloak IdP per l'autenticazione federata e SSO

Università degli Studi di Firenze

Corso di Laurea Magistrale in Ingegneria Informatica

Insegnamento di Software Architectures and Methodologies

Alessandro Lemmo

Sommario

1	Introduzione	3
2	Design.....	3
3	Implementazione	5
3.1	SPI.....	5
3.1.1.	UserStorageProviderFactory	5
3.1.2.	UserStorageProvider	5
3.1.3.	UserLookupProvider.....	5
3.1.4.	UserQueryProvider	6
3.1.5.	CredentialInputValidator	7
3.1.6.	CredentialInputUpdater	7
3.2	Service Layer	8
3.3	DataAccessObject.....	9
3.4	Model.....	10
4	Database.....	11
	Appendice A.....	12

1 Introduzione

Keycloak è una soluzione open source per la gestione dell'identità e degli accessi. L'obiettivo di questo lavoro è quello di implementare un identity provider basato sulla tecnologia keycloak per realizzare l'autenticazione federata. Quest'ultima consiste nell'andare a creare un accordo tra il provider di identità e un'organizzazione, all'interno della quale risiedono i dati degli utenti. Così facendo è possibile accedere ad un servizio effettuando il login per mezzo dell'identity provider (in questo caso keycloak) ma senza avere alcuna informazione memorizzata su di esso, bensì in un database esterno. Operando in questo modo si realizza con semplicità il SSO (Single Sign On) ovvero il fatto di poter accedere a più servizi effettuando una sola volta l'autenticazione e mantenendo i propri dati solamente all'interno della propria organizzazione.

2 Design

L'identity provider realizzato può essere utilizzato all'interno del contesto di un'applicazione in architettura SOA (Service Oriented Application), per far sì che l'accesso a differenti servizi possa avvenire in modalità federata e Single Sign On.

L'applicazione, come riportato nel diagramma UML semplificato alla pagina seguente, è costituita dai seguenti package.

- ***spi***: implementa le Service Provider Interfaces di keycloak, in particolare *UserStorageProviderFactory* realizza *customUserStorageProviderFactory* mentre *UserStorageProvider* implementata una serie di spi che sono illustrate in dettaglio nel capitolo successivo.
- ***service***: contiene la classe *UserService* il quale è un livello intermedio tra le spi ed i Data Access Object. Si occupa di fare tutte quelle operazioni preliminari alla manipolazione dei dati sul database.
- ***dao***: contiene la classe *UserDAO* che espone i metodi atti alla modifica del database. Per fare ciò ci si avvale dell'implementazione JPA di Hibernate.
- ***model***: contiene la classe *User* che è l'entità mappata sul database tramite annotazioni JPA. È inoltre presente la classe *UserAdapter* la quale si occupa di aggiungere informazioni ad un'istanza di *User*. Questa è utilizzata

nei metodi di *customUserStorageProvider* in quanto è necessario ritornare un utente con relativa sessione e realm di appartenenza.

- **log:** contiene la classe *Logger* la quale si occupa di mostrare nella console dei messaggi di log relativi alle operazioni eseguite.

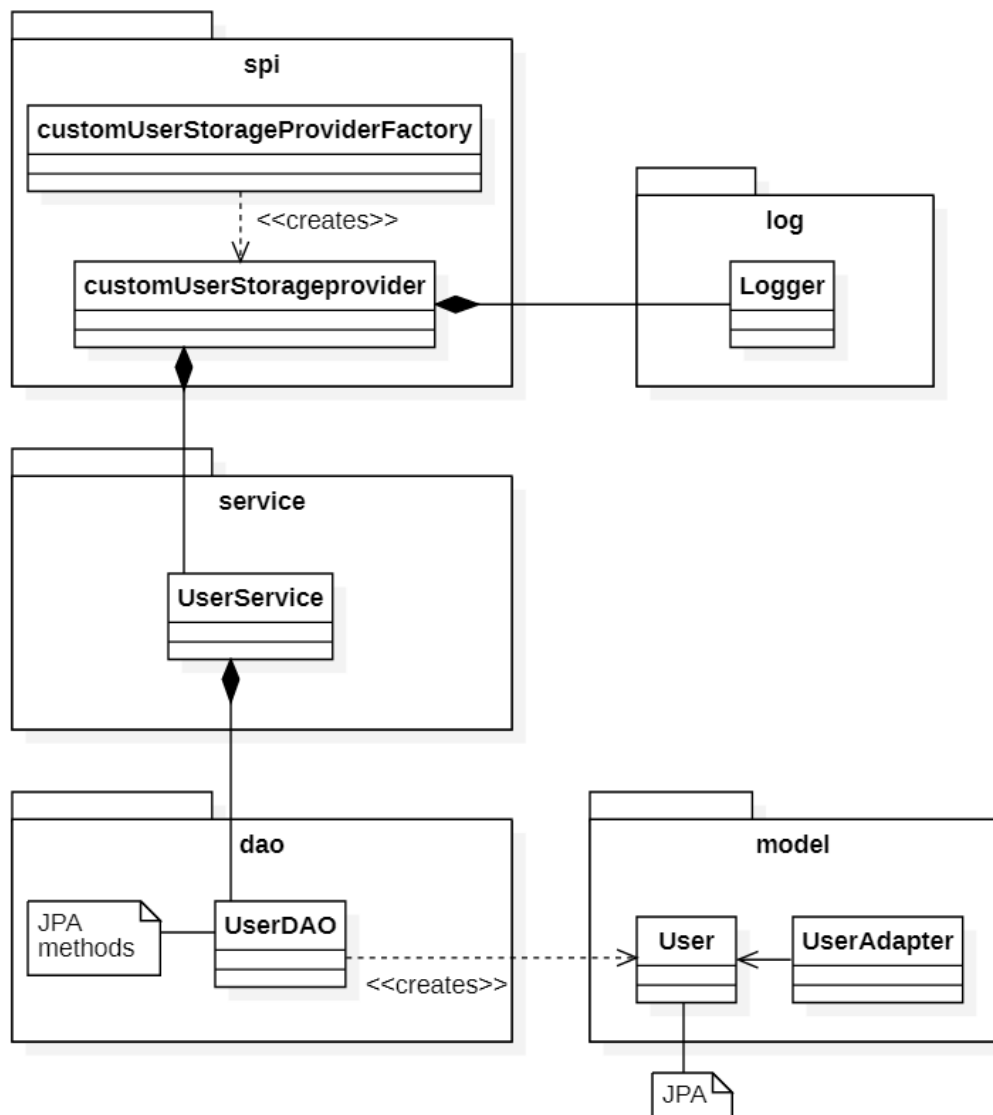


Figura 1 - schema UML semplificato dell'applicazione realizzata.

Uno schema UML completo è fornito in Appendice A.

3 Implementazione

3.1 SPI

Di seguito sono riportate tutte le Service Provider Interfaces implementate con i relativi metodi.

3.1.1. UserStorageProviderFactory

È un'interfaccia composta da undici metodi della quale sono stati implementati i seguenti:

- **getId():** restituisce un identificatore univoco per il provider implementato che Keycloak mostrerà nella sua pagina di amministrazione.
- **create(KeycloakSession session, ComponentModel model):** restituisce l'effettiva implementazione del provider. Keycloak invoca tale metodo per ogni transazione, ovvero per qualsiasi azione che richieda l'accesso all'archivio utente, passando un *KeycloakSession* e un *ComponentModel* come argomenti.

3.1.2. UserStorageProvider

Questa interfaccia contiene solo metodi molto semplici per manipolare gli utenti, nessuno dei quali si riferisce ad utenti effettivi. Tuttavia, le capacità del provider di archiviazione vengono estese implementando le interfacce illustrate di seguito, in modo da poter gestire funzionalità specifiche.

3.1.3. UserLookupProvider

Questa interfaccia è usata per recuperare dal database gli utenti ed importarli all'interno di keycloak. Implementa i seguenti metodi:

- **getUserById(String id, RealmModel realm)**
- **getUserByUsername(String email, RealmModel realm)**
- **getUserByEmail(String email, RealmModel realm)**

Tutti e tre i metodi prendono in input il realm e il valore *String* da cercare. Demandano poi a *UserService* e a sua volta a *UserDAO* una query sul database per recuperare l'utente cercato. Tramite la classe *UserAdapter* effettuano una

mappatura dell'entità recuperata dal database in un *UserModel*. Quest'ultima è la classe che rappresenta un utente come previsto da Keycloak.

3.1.4. UserQueryProvider

Questa interfaccia fornisce una serie di metodi per la visualizzazione nella console di amministrazione di keycloak degli utenti importati da un database esterno. I metodi implementati sono i seguenti:

- **getUsersCount(RealmModel realm)**: si occupa di recuperare il numero complessivo di utenti nel realm delegando la richiesta a *UserService*. Ritorna un valore intero.

Si hanno poi una serie di metodi *getXXX* e *searchXXX* atti alla restituzione di utenti sulla base di diversi attributi passati come parametri. Sono stati implementati i seguenti:

- **getUsers(RealmModel realm)**: demanda il recupero di tutti gli utenti presenti nel database appartenenti al realm a *UserService* e ritorna le entità ottenute mappandole in *UserModel*
- **getUsers(RealmModel realm, int firstResult, int maxResults)**: analogo al metodo sopra ma aggiunge i due parametri int per limitare i risultati.
- **searchForUser(String search, RealmModel realm)**: demanda a *UserService* il recupero delle entità tramite il parametro search e ritorna le entità ottenute mappandole in *UserModel*.
- **searchForUser(String search, RealmModel realm, int firstResult, int maxResults)**: analogo al metodo sopra ma aggiunge i due parametri int per limitare i risultati ritornati.
- **searchForUser(Map<String, String> params, RealmModel realm)**: consente la ricerca tramite una mappa di attributi chiave/valore. Delega a *UserService* la ricerca tramite mappa e realm e ritorna le entità ottenute mappandole in *UserModel*
- **searchForUser(Map<String, String> params, RealmModel realm, int firstResult, int maxResults)**: analogo al metodo sopra ma aggiunge i due parametri int per limitare i risultati ritornati.

- **getGroupMembers(RealmModel realm, GroupModel group)**: permette di ottenere tutti gli utenti appartenenti ad uno stesso gruppo. Delega a *UserService* la ricerca tramite realm e group e ritorna le entità ottenute mappandole in *UserModel*.
- **getGroupMembers(RealmModel realm, GroupModel group, int firstResult, int maxResults)**: analogo al metodo sopra ma aggiunge i due parametri int per limitare i risultati ritornati.
- **searchForUserByAttribute(String attrName, String attrValue, RealmModel realm)**: consente di effettuare la ricerca di un utente tramite una singola coppia chiave/valore. Delega a *UserService* la ricerca e ritorna i risultati mappandoli in *UserModel*.

3.1.5. CredentialInputValidator

Questa interfaccia fornisce una serie di metodi per la convalida delle credenziali utente:

- **supportsCredentialType(String credentialType)**: siccome Keycloak supporta diversi tipi di credenziali, il provider deve specificare quali di queste può gestire. Nel nostro caso solamente le password.
- **isConfiguredFor(RealmModel realm, UserModel user, String credentialType)**: il provider, nell'ambito di un determinato realm, viene configurato per il supporto a determinate credenziali. Essendo nel nostro caso solo le password, tale metodo richiama *supportsCredentialType*.
- **isValid(RealmModel realm, UserModel user, CredentialInput credential)**: verifica se *credential* è di un tipo supportato e successivamente delega a *UserService* la verifica delle credenziali inserite.

3.1.6. CredentialInputUpdater

Fornisce una serie di funzionalità per l'aggiornamento delle credenziali attraverso la console utente. I metodi implementati sono i seguenti:

- **supportsCredentialType(String credentialType)**: analogo all'omonimo metodo in *CredentialInputValidator*.
- **updateCredential(RealmModel realm, UserModel userModel, CredentialInput input)**: verifica che *input* sia tra i tipi di credenziali supportate e delega

l'aggiornamento di *userModel* alla classe *UserService*. Restituisce un valore booleano per indicare se l'operazione è andata a buon fine.

- **disableCredential(RealmModel realm, UserModel userModel, String credential)**: verifica se *credential* è tra i tipi supportati e delega l'aggiornamento di *userModel* alla classe *UserService*.

3.2 Service Layer

È un livello che fa da intermediario tra la classe *UserStorageProvider* e *UserDAO*. Il livello DAO dovrebbe occuparsi solamente delle operazioni sul database e pertanto tutte le azioni necessarie sui dati prima di inserirli, cercarli o eliminarli dal database vengono fatte in questo livello. Pertanto, ogni metodo della classe *customUserStorageProvider* effettua una chiamata ad un metodo di *UserService* che effettua delle operazioni (se necessarie) e successivamente chiama un metodo della classe *UserDAO* per persistere le operazioni.

Di seguito sono riportati tutti quei metodi di *UserService* che effettuano una o più operazioni sui dati prima di passare a *UserDAO* la richiesta per essere persistita sul database. I metodi che si limitano ad inoltrare la richiesta verso *UserDAO* non sono riportati.

- **findUsersByParams(Map<String,String> params, RealmModel realm)**: usa il metodo privato *prepareMap(params)* per creare una mappa adeguata per l'esecuzione della query, ovvero dove ci siano tutti i campi attesi in modo ordinato ed eventualmente nulli se non presenti nella mappa iniziale, e delega a *UserDAO* l'ottenimento dei risultati. Ritorna una lista di *User*.
- **findUsersByParams(Map<String,String> params, RealmModel realm, int firstResult, int maxResults)**: analogo al metodo sopra ma limita i risultati ottenuti per mezzo dei parametri interi *firstResult* e *maxResults*.
- **updateCredentials(RealmModel realm, UserModel user, CredentialInput input)**: delega a *UserDAO* la ricerca dell'utente tramite *realm* e *user*, successivamente aggiorna la password dell'utente ottenuto e sempre tramite *UserDAO* persiste l'aggiornamento dell'entità. Ritorna un valore booleano per indicare se l'operazione è andata a buon fine.

- **disableCredential(RealmModel realm, UserModel user):** delega a *UserDAO* la ricerca dell'utente tramite *realm* e *user*, successivamente setta la password dell'utente ottenuto come *null* e sempre tramite *UserDAO* persiste l'aggiornamento dell'entità.
- **validateCredentials(RealmModel realm, UserModel user, CredentialInput input):** demanda a *UserDAO* la ricerca di un utente tramite i parametri *realm* e *user* e successivamente verifica se la password dell'utente recuperato coincide con *input*. Ritorna un valore booleano per indicare se le credenziali sono valide.

3.3 DataAccessObject

È implementato dalla classe *UserDAO*, ed espone tutti i metodi atti alla modifica del database.

- **getAllUsers(String realmName):** ottiene tutti gli utenti di un realm eseguendo la query *getAllUsers* della classe *User*. Ritorna una list di *User*.
- **getAllusers(String realmName, int firstResult, int maxResults):** analogo al metodo precedente ma limita i risultati ottenuti settando i parametri della query *firstResult* e *maxResults*.
- **getUsersCount(String realmName):** ottiene il numero totale di utenti appartenenti ad un realm eseguendo la query *getUsersSize* della classe *User* e ritorna l'intero risultante.
- **findUserById(String id, String realmName):** esegue la query *getUserById* della classe *User*, per ottenere l'utente con identificativo *id*.
- **findUserByUsername(String username, String realmName):** esegue la query *getUserByUsername* della classe *User*, per ottenere l'entità con nome utente *username*.
- **findUserByEmail(String email, String realmName):** esegue la query *getUserByEmail* della classe *User*, per ottenere l'entità con l'email passata.
- **findUsers(String search, String realmName):** esegue la query *SearchForUser* per ritornare una lista di *User* che contengono in uno dei campi la stringa *search*.

- **findUsers(String search, String realmName, int firstResult, int maxResults)**: analogo al metodo precedente ma limita i risultati ottenuti settando i parametri della query *firstResult* e *maxResults*.
- **findUsersByParams(Map<String,String> params, String realmName)**: scorre la mappa chiave/valore ed esegue la query *SearchForUserByParams* settando per ogni attributo (chiave) il corrispondente dato (valore).
- **findUsersByParams(Map<String,String> params, String realmName, int firstResult, int maxResults)**: analogo al metodo precedente ma limita i risultati ottenuti settando i parametri della query *firstResult* e *maxResults*.
- **findUsersByGroup(String realmName, String groupName)**: esegue la query *SearchForUsersByGroup* e ritorna una lista contenente tutti gli *User* appartenenti ad uno stesso gruppo.
- **findUsersByGroup(String realmName, String groupName, int firstResult, int maxResults)**: analogo al metodo precedente ma limita i risultati ottenuti settando i parametri della query *firstResult* e *maxResults*.
- **findUserByAttribute(String attrName, String attrValue, String realmName)**: crea una query che seleziona tutti gli *User* con attributo *attrName* con valore *attrValue*. Ritorna una lista di utenti.
- **updateCredentials(User entity)**: esegue all'interno di una transazione il merge dell'entità aggiornata *entity*.
- **disableCredential(User entity)**: esegue all'interno di una transazione il merge dell'entità aggiornata *entity*.

3.4 Model

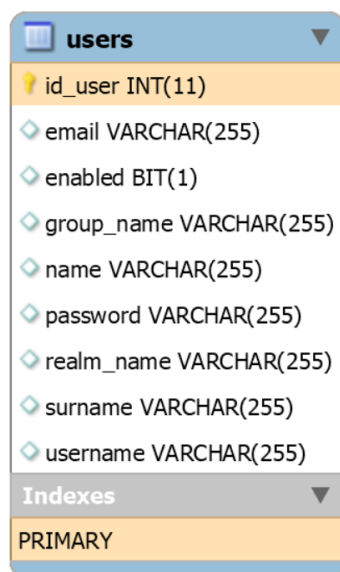
Contiene la classe *User*, la quale realizza tramite le annotazioni JPA l'entità che è mappata sul database. Tale classe contiene una serie di attributi relativi ad un'istanza di *User* ed una serie di *@NamedQuery* che sono riportate di seguito:

- **getAllUsers**: seleziona tutti gli utenti di un realm dalla tabella *User*.
- **getUserById**: seleziona l'utente di un realm con un dato id.
- **getUserByUsername**: seleziona l'utente di un realm con un dato username.
- **getUserByEmail**: seleziona l'utente di un realm con una data email.

- **getUsersSize**: ritorna il numero complessivo di utenti in un realm
- **searchForUser**: seleziona tutti quegli utenti che hanno nello username o nell'email una data stringa.
- **searchForUsersByParams**: seleziona tutti quegli utenti che hanno name, surname, email, username concordi con i valori dati, i quali possono essere anche nulli.
- **searchForUsersByGroup**: seleziona tutti gli utenti di un realm appartenenti ad uno stesso gruppo dato.

4 Database

Il database utilizzato per il salvataggio degli utenti è stato implementato con MySQL. È costituito da una sola tabella “Users” avente i campi riportati nell'immagine seguente.



users	
id_user	INT(11)
email	VARCHAR(255)
enabled	BIT(1)
group_name	VARCHAR(255)
name	VARCHAR(255)
password	VARCHAR(255)
realm_name	VARCHAR(255)
surname	VARCHAR(255)
username	VARCHAR(255)
Indexes	
PRIMARY	

Figura 2 - Tabella degli utenti con i relativi attributi.

Appendice A

