Will Passidomo

OfficeShare API(stage 3) Design Document
AuthenticationAPI
11/20/14

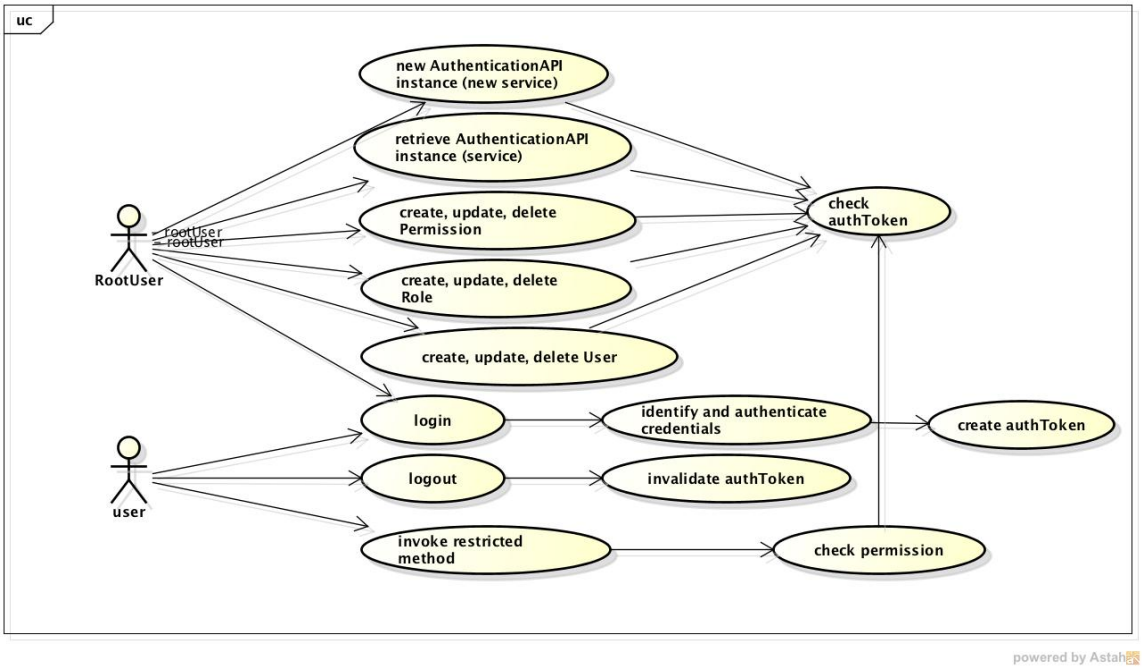
Introduction

This document is meant to be used as a design for AuthenticationAPI, which is a modular service which provides services for restricting access to protected methods and tracking sessions and user credentials. As it pertains to this document, this service is being used in conjunction with RenterAPI(see implementation note), a module of the Squaredesk application.

Use Cases

AuthenticationAPI supports two different Use Cases.

1) A root user
    a. is able to call CRUD on new Instances of the AuthenticationAPI, each for a different service
    b. is able to lend its credentials to a service (the actual program) to dynamically call CRUD methods on Users, Roles, Permissions, and AuthTokens
    c. is able to, itself, to call CRUD methods on Users, Roles, Permissions, and AuthTokens
2) User
    a. Is able to login and gain a valid authToken
    b. Is able to logout and invalidate a authToken
    c. Have permissions and invoke restricted methods provided they have the permission in question



Overview

The value of the Squaredesk Application, reviewed in the previous two design documents for OfficeSpaceProviderAPI and RenterAPI, is only possible with access control and basic security features. In order to keep the program from breaking and to dissuade malicious attacks or identity theft, the presence and integration of an Authentication service is necessary. For this purpose, we have set out to design a service called AuthenticationAPI.

While AuthenticationAPI is built for the Squaredesk Application, it has been designed so that it could effectively serve any application. AuthenticationAPI takes the responsibility for validating login credentials, tracking sessions and validating access for restricted methods from the service which employs it.

AuthenticationAPI offers an interface which the Root User representing a service may access and create, read and update their whole set of Permissions, Roles, Users pertaining to their service. AuthenticationAPI strives to provide a safe, secure and easy to use program for developers to intergrate in a wide range of applications.

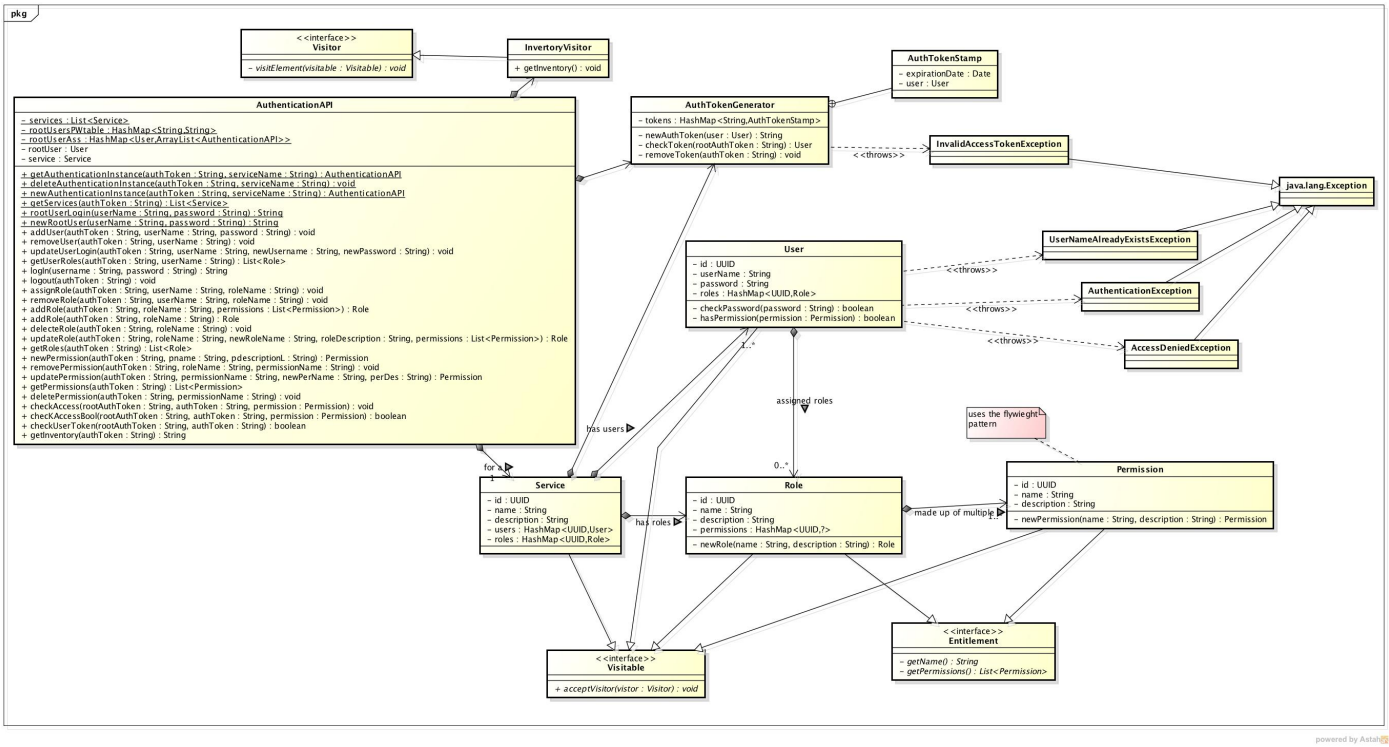
Requirements

1) to support the creation, update and deletion of Users. Users are a basic entities which represent Users in service which is employing the AuthenticationAPI
2) to support the creation, update and deletion of Roles and Permissions, which both implement Entitlements. The tracking of User's associations with Permissions is one of the major functional elements of the AuthenticationAPI. Permissions are organized into sets, in Roles. Roles

may contain permissions or other Roles, which makes it proper to categorize them both as Entitlements. Roles must be easily assigned to Users

3) to support the create, update and deletion of services, a term synonomus with each separate instance of the AuthenticationAPI. Services must belong to a RootUser and a RootUser may own multiple services. All User, Role, and Permission data must belong to one and only one service
4) to support the creation, update, deletion and tracking of authTokens
5) to employ the visitor patter to implement a method called getInventory() which visits each User, Role, and Permission and compiles and Inventory of active items

Class Directory



AuthenticationAPI

The AuthenticationAPI is used to track and validate user's permissions for access. The AuthenticationAPI may be extended for any service by creating a new instance for the new service. Each instance tracks the users of a service, the defined roles and which roles each user has been assigned, the defined permissions within each role and valid authTokens. AuthenticationAPI provides methods for updating the assignment of permissions and roles, creating and deleting users, roles, and

permissions and creating new authTokens. This class also supports the printing of an inventory of all of its associations by employing the Visitor Pattern.

In order to create a new instance to user AuthenticationAPI for your service, You must first register yourself as a Root User by calling the static newRootUser(String, String, UserDTO) method. If this returns true, then you have successfully created a root user account, it returns false, it means you have selected a duplicate username. Once you have created your Root User account, you must call RootUserLogin(String, String) which will return an AuthToken which is good for the length of your session.

To create a new service, you may call the static method newAuthenticationAPIInstance(String, String), passing in your authToken and the name of your service as parameters. The program will automatically check that you have root user permission and create the new service instance. To retrieve this instance, you will pass the same information to getAuthenticationAPIInstance(String, String), which will return the instance. Instances may only have one Root User who is able to access the instance.

Static Method Name	Signature	Description
getAuthenticationInstance	(String authToken, String serviceName): AuthenticationAPI	Checks authToken for validity, and returns the instance of AuthenticationAPI which corresponds to the serviceName parameter
deleteAuthenticationInstance	(String authToken, String serviceName): void	Checks authToken for validity, and removes the instance of AuthenticationAPI which corresponds to the serviceName parameter
newAuthenticationInstance	(String authToken, String serviceName, String serviceDescription): AuthenticationAPI	Checks authToken for validity and creates a new instance of AuthenticationAPI with a new Service with the corresponding name and description to the serviceName parameter and serviceDescription parameter
getServices	(String authToken): List<Service>	Returns a list of all services the AuthenticationAPI is being used for (all the different instances)
rootUserLogin	(String username,	Method for a rootUser of an

	String password):String	AuthenticationAPI instance to login and return an authToken which them may then use to fetch 1 or more of the AuthenticationAPI's for which they are root users
newRootUser	(String username, String password):String	Create a new rootuser login, returns an authToken which may be used to call newAuthenticationInstance()

Method Name	Signature	Description
addUser	(String authToken, String login, String password):User	Checks authToken for validity as RootUser. Adds new User to the service. New User has no permissions as default
removeUser	(String authToken, UUID userID):void	Checks authToken for validity as RootUser. Removes the specified user
updateUserLogin	(String authToken, String username, String password): void	Checks authToken for validity as RootUser. Replaces user's username and password
getUsersRoles	(String authToken, String username, String roleName): List<Role>	Checks authToken for validity as RootUser, returns the roles associated with the User with username, userName
getUsers	(String authToken) : List<User>	Checks authToken for validity as RootUser, returns a list of all Users associated with service
Login	(String authToken, String username, String password):String	Checks authToken for validity as RootUser, checks username and password for match in service's user table, returns authToken for User if found, raises AuthenticationException if not found
Logout	(String rootAuthToken, String usrAuthToken) :void	Checks rootAuthToken for validity as RootUser, checks usrAuthToken for validity as service user token, then removes usrAuthToken from

		valid Tokens.
assignRole	(String authToken, String userName, String roleName):void	Checks authToken for validity as RootUser, adds Role instance corresponding with roleName parameter to User corresponding with userName's List of Roles
removeRole	(String authToken, String userID, Stiring roleID):void	Checks authToken for validity as RootUser, removes the Role instance corresponding with roleName parameter from User corresponding with userName's List of Roles
addRole	(String authToken, String roleName, String roleDescription, List<Entitlement> entitlements): Role	Checks authToken for validity as RootUser, Adds a new Role object with the Entitlements listed in the entitlements parameter to the service. Returns the role added
addRole	(String authToken, String roleName, String roleDescription): Role	Checks authToken for validity as RootUser, Adds a new Role object with no permission to the service. Returns the role Added
deleteRole	(String authToken, String roleID) :void	Checks authToken for validity as RootUser, deletes the Role with the specified UUID in the roleID parameter
getRoles	(String authToken): List<Role>	Checks authToken for validity as RootUser, returns a list of all Roles associated with service
updateRole	(String authToken, String roleName, String newRoleName, String description, List<Entitlement> entitlements):Role	Checks authToken for validity as RootUser, replaces the name, description and entitlements in the Role instance corresponding with roleName with the parameters, newRoleName, description, and entitlements. Returns the update Role object
addPermissionToRole	(String authToken, String roleName, String permissionName):	Checks authToken for validity as RootUser, adds Entitlement associated with

		name permissionName to the Role associated with roleName
getRolesPermissions	(String authToken, String roleName): List<Role>	Checks authToken for validity as RootUser, returns the Permissions associated with the Role with name, roleName
removeEntitlement	(String authToken, String roleName, String entitlementName):Void	Checks authToken for validity as RootUser., removes Entitlement associated with entitlementName from role associated with roleName
newPermission	(String authToken, String permissionName, String permissionDescrip): Permission	Checks authToken for validity as RootUser, creates a new Permission object with the name, permissionName and with the description permissionDescrip. Returns the new Permission
deletePermission	(String authToken, String permissionName):void	Checks authToken for validity as RootUser, removes the permission associated with the name permissionName.
updatePermission	(String authToken, String permissionName, String newPerName, String description): Permission	Checks authToken for validity as RootUser, replaces the name, description and permissions in the Permission instance corresponding with permissionName with the parameters, newPerName, description, and permissions. Returns the update Role object
getPermissions	(String authToken): List<Permission>	Checks authToken for validity as RootUser, returns a list of all Permissions associated with service
checkAccess	(String rootAuthToken, String authToken, String permission): void	Checks authToken for validity as RootUser, asses authToken as valid User token if valid, gains reference to user and tests whether User has a Permission matching

		permission. Throws InvalidAccessTokenException if authToken or rootAuthToken is invalid. Throws AccessDeniedException if User does not have Permission matching permission
checkAccessBOOL	(String rootAuthToken, String authToken, String permission):boolean	Similar to checkAccess, but does not throw Exception, rather returns boolean value. Checks authToken for validity as RootUser, gains reference to user and tests whether User has a Permission matching permission. Returns true if authToken and rootAuthToken are valid and User has Permission matching permission, returns false if any of the above is not true
getInventory	(String authToken): String	Checks authToken for validity as RootUser, retruns String representation of all Users of service and their Roles, all Roles (w/description) of service and their Permissions and all Permissions (w/description) of service

Static Associations	Type	Description
services	List<Service>	A list of all services
rootUsersPWtable	HashMap<String,String>	A username, hashed password map, used to validate root users logins
rootUserAss	HashMap<User, ArrayList<AuthenticationAPI>>	A Map used to track which AuthenticationAPI instances belong to which root user.
apiTokenGen	AuthTokenGenerator	The AuthTokenGenerator

		instance which tracks, provides and validates authTokens for Root Users

Associations	Type	Description
rootUser	User	The root user for the given instance
service	Service	The application who owns the instance

User implements- Visitable

The user class represents a User in a service. Users are assigned roles (containing permissions) which correspond to the methods which the root user would like to allow the user to perform within the service. Users have 4 associations, and id, a username, a password and 0 or more Roles. User's have 2 methods, which are called internally, checkPassword(String) which return's true if the hashed password provided matches the stored hashed password, and hasPermission(Permission), which returns true if the user has been assigned the passed permission.

Property Name	Type	Description
Id	UUID	The unique id for the User
userName	String	The username for the user
password	String	The users hashed password

Association Name	Type	Description
Roles	HashMap<String, Role>	A map of the users assigned Role's names and the Role object

Method Name	Signature	Description
checkPassword	(String password):boolean	Checks if the provided hashed password matches the stored hashed password
hasPermission	(String permission):boolean	Iterates through the users roles and checks to see if the any of them match the permission provided. Throws AccessDeniedException if

		Permission corresponding with permission parameter is not found

Service implements- Visitable

The Service class represents a service which is associated with an instance of AuthenticationAPI. Service has 1 method, the abstract acceptVisitor(Visitor). Service has 4 fields, id, name, description and a HashMap representing the User accounts associated with the service.

Property Name	Type	Description
id	UUID	The unique id for
name	String	A name for the Service
description	String	A description of the service

Association Name	Type	Description
roles	HashMap<String, Role>	A map of the Role's names associated with a Service instance to the Role object
users	HashMap<String, User>	A map of the User's usernames associated with the Service instance to the User object
permissions	HashMap<String, Permission>	A map of the Permission's names associated with the Service instance to the Permission object
tokenGen	AuthTokenGenerator	The AuthTokenGenerator which tracks, provides and validates authTokens for Users of the Service instance

Static Association Name	Type	Description
services	HashMap<String, Service>	A map of the Service's names instantiated by AuthenticationAPI mapped to the Service object

Role implements- Visitable, Entitlement

The Role class represents a classification of a set of Permissions. The permissions within a role usually constitute the functionality needed to fulfill the abilities of one of the actors in a Use case within an actual service. Role has 4 associations, an id, a name, a description and a HashMap representing the Permision Instances associated with the service. Role has 2 methods, the implementation of the abstract acceptVisitor(Visitor) and newRole(String, String) which uses the Flyweight pattern to construct new Role objects or return existing Role objects

Association Name	Type	Description
id	UUID	The unique id for the Entitlement
name	String	A name for the Entitlement
description	String	A description of the Entitlement

Association Name	Type	Description
Permissions	HashMap<String,Entitlement>	A map of the Entitlement' names associated with a Role instance to the Entitlement Object

Permission implements- Visitable, Entitlement

The Permission class represents a single permission which should correspond to a requirement that a "user" of a service must be granted said permission to access an otherwise restricted method piece of data. Permission has 3 associations, an id, a name and a description. Permission has 2 methods, the implementation of the abstract acceptVisitor(Visitor) and newPermission(String, String) which uses the Flyweight pattern to construc new Permission objects or return existing Permission objects.

Association Name	Type	Description
id	UUID	The unique id for the Entitlement
name	String	A name for the Entitlement
description	String	A description of the Entitlement

Entitlement(interface)

Entitlement is the interface implemented in Permission and Role. Entitlement has no methods and is mainly functional as the extension of a class which may be added to a Role. Entitlement's only method getPermissions() is used

to return either the permissions of a role or the single permission of a Permission object

Method Name	Signature	Description
getPermissions	():List<Permission>	Returns the Permissions contained within a role, if implement in Role or the Permission itself if implemented in a Permission instance

AuthTokenGenerator

Is a class for creating new AuthTokens, for testing validity of submitted tokens and for matching token with the User it belongs to. AuthToken's are String representations of UUID objects, a universally Unique 128 bit value. When a new authToken is generated, it is put in a HashMap which maps the String value of the Token to an AuthTokenStamp object. AuthTokenStamp is an inner class of AuthTokenGenerator which is responsibility for logging the time a stamp is set to expire and the user it was assigned to.

Association Name	Type	Description
tokens	HashMap<String,AuthTokenStamp>	A map of tokens

Method name	Signature	description
newAuthToken	(User user):String	Returns a new, valid authtoken mapped to the User, user
checkToken	(String authToken):User	Returns the User associated with the token if it is valid,. Throws InvaliAccessTokenException if token is not mapped or if token has expired.
removeToken	(String authToken):void	Removes the passed token, authToken from the map. Throws InvalidAccessTokenException if token is not mapped.

AuthTokenStamp (inner class)

AuthTokenStamp is an Immutable class with only 2 properties, a Date, expirationDate and a User, user. AuthTokenStamps are created by AuthTokenGenerator, each to track a particular authToken. Every authToken

maps to an instance of AuthTokenStamp in AuthTokenGenerator HashTable "tokens"

Property Name	Type	Description
user	User	The User instance associated with the authToken
expirationDate	GregorianCalendar	The expiration date, set to 24 hours ahead of the instance creation date

InventoryVisitor implements Visitor

ImplementsVisitor (a class designed in accordance with the basic Visitor pattern). InventoryVisitor uses its method getInventory to Iterate through the list of Users of the service, their Roles and their Entitlements and returns a clean, list in String format.

Method Name	Signature	Description
getInventory	():String	Returns an organized Inventory of all Users, Roles, Entitlements, Permissions and Tokens

AuthenticationException extends Exception

AuthenticationException is a marker class which is thrown by the getUser(String username, String password) method of a Service class instance. AuthenticationException signifies that the credentials supplied are not valid, either the Username provided does not exist or the Password does not exist. Taking into account where AuthenticationExceptionis thrown, it would be trivial to include a message about which one of the two (username or password) was incorrect, but due to security concerns, the difference is not noted, rather the more generic message "Incorrect Username and/or password" is displayed

Property Name	Type	Description
userName	String	The username attempted to login with
password	String	The password attempted to login with

AccessDeniedException extends Exception

AccessDeniedException is a marker class which is thrown by the hasPermission(String permission) method in a User instance.

AccessDeniedException signifies that the User has not been assigned the Permission specified in the parameter. AccessDeniedException displays a message stating the (User's username) does not have (Permission's name) permission

Property Name	Type	Description
permission	String	The permission which the user was tested against and didn't have

Association Name	Type	Description
user	User	The user for whom access was denied

InvalidAccessTokenException extends Exception

Invalid AccessTokenException is a marker class which is thrown by the checkToken(String authToken) and the removeToken(String authToken) methods in the AuthTokenGenerator. InvalidTokenException signifies that one of the following is true; the authToken parameter does not match any recorded authToken, the authToken parameter does match a recorded authToken, but that token is expired. The cause for the exception being thrown, out of the two just mentioned, is specified in the message

Association name	Type	Description
user	User	The user who passed the token (when token was expired)

UserNameAlreadyExistsException extends Exception

UserNameAlreadyExistsException is a marker class which is thrown by the newUser(String username, String password) method in the User class. UserNameAlreadyExistsException signifies that the username parameter matches the username, case insensitively, of an existing User instance'

Property name	Type	description
userName	String	The username which was attempted to be duplicatied

Association name	Type	Description
user	User	The User who attempted the name change

service	Service	The service which the user resides and the duplicate naming took place

Implementation Details

Notes: my implementation of OfficeSpaceProvider was woeful at performing its basic functions when I handed it in, and it's lack of a clear public interface made it restrictively difficult to tie in the AuthenticationAPI so that there was any meaningful access control taking place. In lieu of essentially rebuilding the OfficeSpaceProvider from the ground up, I added a newUser() function to RenterAPI (which creates a new User) and only tied RenterAPI into AuthenticationAPI, so any areas, such as the sequence diagram, where it asked to show how RenterAPI, ProviderAPI and AuthenticationAPI worked together, I used RenterAPI in place of both RenterAPI and ProviderAPI. In the testDriver, I retained the script I worte for Assignment 2 which created OfficeSpaces and OfficeSpaceProviders for the sake of testing, but those methods have no access control.

To compile this project, run the following command

javac cscie97/asn4/squaredesk/renter/*.java
cscie97/asn4/squaredesk/provider/*.java
cscie97/asn4/squaredesk/common/*.java
cscie97/asn4/squaredesk/KnowledgeEngine/*.java cscie97/asn4/test/*.java
cscie97/asn4/squaredesk/authentication/*.java
cscie97/asn4/squaredesk/RenterAPI/*.java

AuthenticationAPI was designed to be as extensible and loosely coupled as possible. AuthenticationAPI was originally designed to follow the Singleton pattern, but the real implementation makes use of a Factory and the Flyweight Pattern. That each "service" that uses the Authentication service, is served by a single instance of the AuthenticationAPI class. When a client wishes to fetch their instance, they simply login and call the static getAuthenticationInstance(String authToken, String serviceName) method. The API has 2 types of users. Root users, which have credentials to log into the API and fetch instances and call API methods directly. ServiceUsers (refered to henceforth simply as Users) only exist withing a given service's instance. While Root users are able to use the full functionality of the API, such as assigning, creating and deleting Permissions, Roles, and Users amongst others. Users are not meant to interact directly with the API, rather they mirror

Users or Actors in an actual service and the service uses the AuthenticationAPI by calling the login(String username, String password) which returns credentials which are tracked in the API. Services can use the checkAccessMethod(parameters) to check whether a user has been assigned the Permission which accessed is being check for.

Services can be defined as either a module or a full application which uses the AuthenticationAPI service. In this sprint, the AuthenticationAPI is being used to authenticate requests for the RenterAPI and OfficeSpaceProviderAPI in the Squaredesk application. The original plan was to use two separate instances of the AuthenticationAPI, one for the RenterAPI and one for the OfficeSpaceProviderAPI, but as I began to tie the Authentication services into RenterAPI, it became evident that the interconnectedness of the RenterAPI and the OfficeSpaceProviderAPI, in particular that they shared Users (of the Squaredesk Application) and there were few roles, someone which would have shared permissions, was going to make it so that it would be far more appropriate to treat the Squaredesk Application itself as a "service".

Permissions and authTokens are the backbone of this API. Permissions are simple entities, instances of the Permission class consisting on a simple name property and a description property, both Strings. Permissions are assigned to Users to reflect the entitlement of that User (a separate, but mirrored entity) in the actual service. The naming convention for Permissions Name's is typically the name of the method which tests it, in lowercase, with underscores where spaces typically would be (in typical Java naming conventions, this would be right before an Uppercase letter). For example updateRenterName() would typically have the permission "update_renter_name". To check for a permission, the method must require an authToken in its signature, and pass the authToken and the permission name, usually hardcoded, to checkPermission(String authToken, String permission), which simply checks if the user has been assigned the passed permission. If it has it allows to program to carry on as normal and if it hasn't it throws and IllegalAccessException stating that the User does not have the required permission.

AuthTokens are the primary way which users identify themselves within a program in a secure manner. AuthTokens are generated and passed back to Users and RootUsers when they call their respective login methods. Within the AuthenticationAPI, authTokens are used both to test the validity of the request; that the authToken indeed matches a valid authToken that was generated within the last 24 hours and not invalidated by a call to logout(String authToken), and as a means to gaining reference to the user who passed the authToken. authTokens are stored in a Map which maps the actual string of the authToken to a AuthTokenStamp, in inner class of the AuthTokenGenerator. AuthTokenStamp contains two fields, expirationDate and User, the expirationDate being (rather arbitrarily) 24 hours after it was issued and the User being the User it was issued to. This Map provided invaluable to the checkAccess(String authToken, String permission) method described above. When checkAccess is called, the authToken is passed as a key to the map, if no value is returned an IllegalArgumentException is thrown alerting the caller that the authToken is invalid. If a value is returned, the expirationDate is checked against the current system time, if it is before the system time, an

IllegalArgumentException is thron alerting the caller that the authToken is expired. If a value is returned and the stamp is not expired, the reference to the User is returned and tested as to wether or not it has indeed been assigned the permission in question. If it has, the program finishes with checkAccess() and moves onto the net operation.

To make it simpler to reflect the entitlement of a User of a service, in the API, Sets of Permissions are amalgamated into Roles, which are intended to reflect a Use-cases of the service.

In order to make the AuthenticationAPI work dynamically with the RenterAPI and OfficeSpaceProviderAPI it had to be able to validate the program itself by using the RootUser of the service's credentials. When integrating the AuthenticationAPI to RenterAPI, I found it effective to hardcode a method which logged in the RootUser for SquaredeskAPI (the instance of AuthenticationAPI which has the service "SquaredeskAPI") and returned a valid AuthToken, allowing the program to call methods to assign permissions, create users and redirect the RenterAPI.login(String username, String password) method to the login method within AuthenticationAPI. I added to the createRenterProfile(String authToken, RenterDTO renter) a line in the method which retrieves the (RenterAPI) User which the renter profile would be for and passes it to the AuthenticationAPI method assignRole(String authToken, String username, String roleName) along with the RootUsers authToken and the name of the role "renter" which had already been entered into the AuthenticationAPI's instance with the proper permissions to reflect the functionality required by a Renter. The same principal applied when I added to the newUser() method;

Several methods, previously implemented, were modified to make use of the Proxy Pattern in order to maintain the succinctness between the two programs. The RenterAPI. newUser(String username, String password) creates a user in the Squaredesk Application, and also makes a call to the AuthenticationAPI.newUser(String username, String password) method, creating a User with the same username and password (while also assigning the "basic_user" role). Likewise, the Login(String username, String password) and logout(authToken) make similar use of the Proxy pattern by calling the matching method in AuthenticationAPI.

In one instance my implementation did differ from the original design. When it came time for me to implement the User and UserCredentials, I could not see the need to include both String name and String username fields. Since User, for my implementation is only used internally, I decided to apply Occams Razor and eliminate the UserCredentials class, leaving User to have the properties UUID id, String username, and String password.

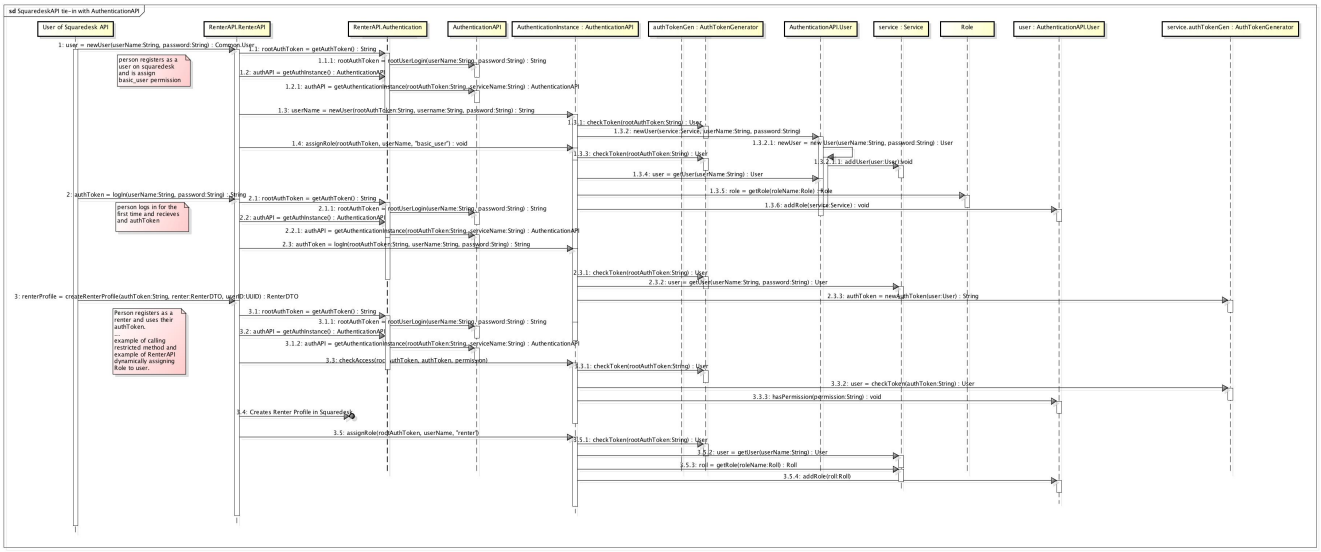
Permissions are assigned to users by the RootUser, typically programmatically, in the form of Roles

AuthenticationAPI uses the visitor pattern to return a string version of the inventory of users, roles, permissions etc

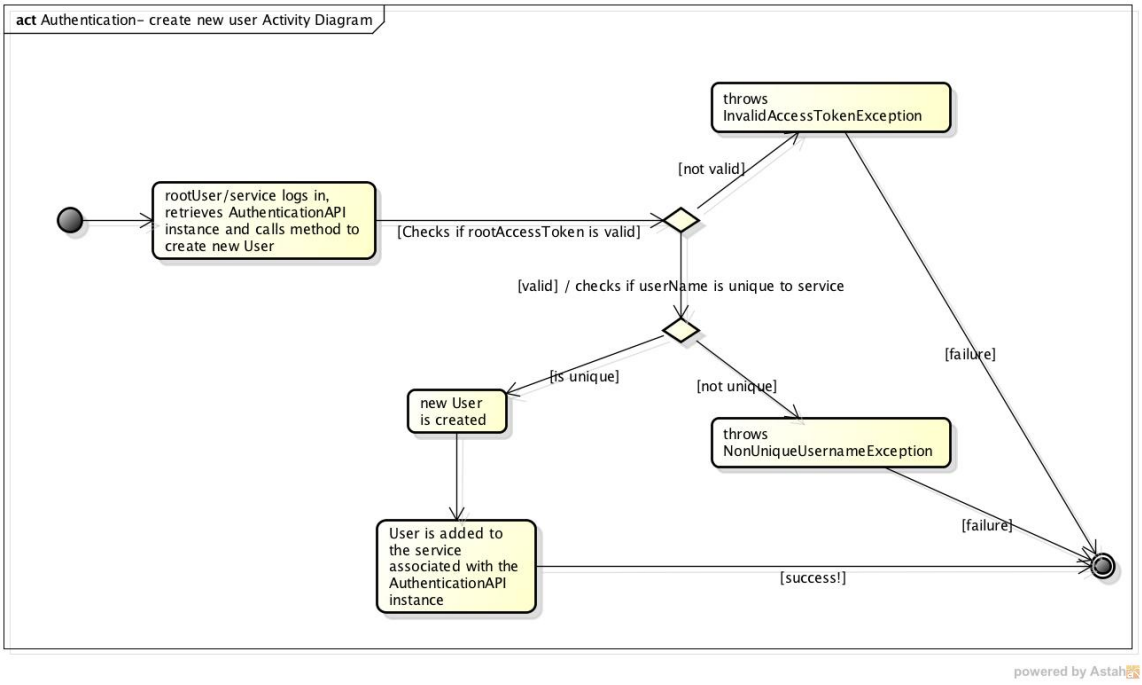
Permission and Role use the Flyweight pattern to generate objects

Sequence Diagram showing the intergration of the RenterAPI and the AuthenticationAPI

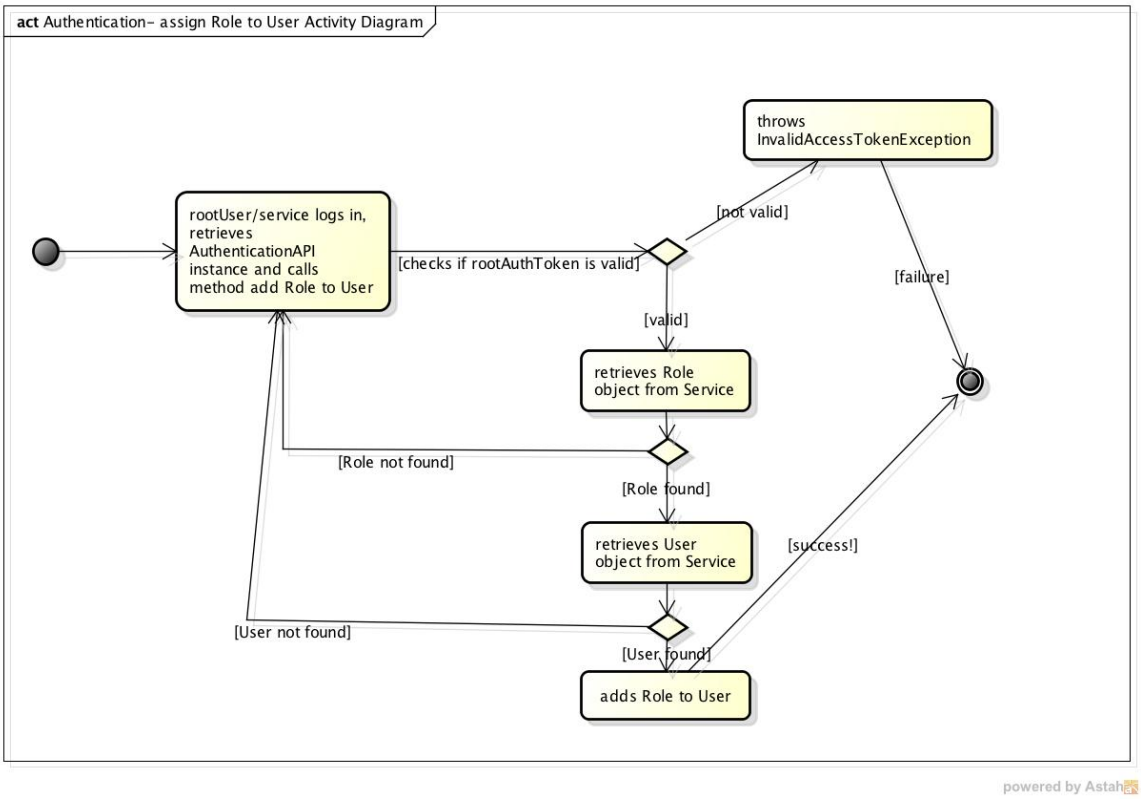
- ***creation of user***
- ***login***
- ***creation of renter profile***



Activity Diagram showing the creation of a new User in the AuthenticationAPI



Activity Diagram showing how a Role is assigned to a User



Testing

In order to test the AuthenticationAPI, run TestDriver and supply the argument “authentication.csv”, as prescribed in the requirements. This will execute an ammalgomation of two programs, a short test driver showing calls made directly to AuthenticationAPI and a rerun of my RenterAPI testDriver, with modifications made to support the intergration of the AuthenticationAPI into the exising program

Through AuthenticationAPI

0) Creates a new RootUser with username “willpassidomo” and password “password”
1) Creates a new Authentication Instance called “SquaredeskAPI”
2) loads “authentication.csv”, creates proper roles, permissions and assigns entitlements to roles
3) calls AuthenticationAPI.getInventory() which retruns the inventory gathered by InventoryVisitor class, and prints inventory

This test program loads a number of OfficeSpaceProviders, OfficeSpace, FacilityTypes, CommonFeatures and PrivateFeatures into the system to act as “seed” elements since they cannot be loaded through the RenterAPI. Then executes the following steps through the renter API:

Through RenterAPI (in testLoader.initializeRenters)

0) create User, adminUser, login with rootUser and assign adminUser "admin" privileges;
1) create user10, user11, user12
2) user10 logs in a creates Renter1;
3) user11 logs in a creates created Renter2
4) user12 logs in a creates created Renter3
5) admin retrieves List of all Renters;
6) deleted Renter3
7) update Name, Contact, PayPal, and Image information of renter2;
8) retrieved Renter1 based on UUID (of created Renter)
9) retrieved a list of all facilityTypes;
10) retrieved a list of all privateFeatures;
11) retrieved a list of all commonFeatures;
12) Renter1 performs a blank query to return all office spaces
13) Renter2 performs a query based on specific criteria
14) Renter 1 makes a reservation for a space;
15) Renter2 makes another reservation for a space;
16) Renter2 cancels his reservation;

And finally

Through AuthenticationAPI

17) calls AuthenticationAPI.getInventory and prints inventory.

ErrorTesting

To desmonstrate the error handling in the AuthenticationAPI. I devised a short program which can be run by executing

java -cp . cscie97.asn4.test.ErrorTest authentication.csv

this program loads up the authentication.csv file (to seed the program) and runs the following script (which will return the indicated error messages)

0) Creates a newRootUser with the following username and password "willpassidomo", "password"
1) Creates a newRootUser with the following username and password "willpassidomo", "123456"
   a. will throw a NonUniqueUserNameException and second user will not be created
2) AuthenticationAPI instance with service named"SquaredeskAuthAPI will be created
3) creates a user with the following username and password, "lebronJames", "secret"
4) creates a user with the following username and password,

"admin", "adminPassword"

5) creates another user with following username and passoword "lebronJames", "password"
   a. will throw a NonUniqueUserNameException and second user will not be created
   b. System.out.println(authAPI.printInventory()); will show 2 users
6) assigns "admin" role to admin
7) assigns "basic_user" role to "lebronJames"
8) user, "lebronJames" signs in and receives and authToken
9) "lebronJames" calls authenticationAPI.checkPermission(string authToken, String permission) passing "get_rentert_list as the permission, to test if the account has access to the corresponding method RenterAPI.getRenterList()
   a. will throw a AccessDeniedException stating lebronJames does not have the require permission for "get_renter_list". This is an unchecked exception in the AuthenticationAPI as it is a void method which relys on throwing the exception to the client to check in order to keep the program running. In this case, this script will catch the error and print the error message and the stack trace.
10) "lebronJames" calls authenticationAPI.checkPermission(string authToken, String permission) passing "get_private_features as the permission, to test if the account has access to the corresponding methodRenterAPI.getPrivateFeatures(), which he does have permission for
11) "lebronJames" signs out, invalidating his authtoken
12) "lebronJames" calls authenticationAPI.checkPermission(string authToken, String permission) passing "get_private_features as the permission, to test if the account has access to the corresponding methodRenterAPI.getPrivateFeatures(), which he does have permission for
   a. will throw an InvalidAccesTokenException because his token is stating his token is no longer valid
13) "lebronJames" attemps to log back in with the password "supersecret"
   a. an AuthenticationException will be thrown saying the Username and of Password is wrong
14) AuthenticationAPI.printInventory is called and printed

Risks

There are 3 major risks that I have identified which pose major risks to the AuthenticationAPI, one of which are inherent to the requiremenmts and two of which, can be attributed to my design and implementation.

1) as per the design, this API does not implement any sort of persistence or database structure. This makes the data inherently vulnerable to loss as well as setting a very low limit as to how much data can be stored, constrained by the limit on active memory

2) the encryption for the passwords seems to be woefully weak. I did not spend much time working on a schema for hashing password, but I did contain it all into one private method, hashPassword(String password): String, which may be easily modified. I hashed all passwords by calling the String method .getHash() which I figured would be an encryption. It was, but it seems as though “password1”, “password2” and “password3” resulted in sequential values. This would not be good for hackers trying to guess passwords

3) There was a bug with the authTokens that I caught in time to report but not to implement a fix. I realized while testing that the design allows for users to log in multiple times and have multiple valid authTokens out at the same time. This was my original plan, as I wanted to leave it easier for distributed applications using the AuthenticationAPI to gain a valid authToken, as long as they had the User’s username and password. Unfortunatly, I did not account for this when designing the logout method, because the logout method only logs out one of the Users valid authTokens. For this to be effective, and truly log the user out, it would need to invalidate all the users authTokens. There are two ways to fix this I can think of.

    1) Add a requirement that Users may only have one authToken out at a time and apply the flyweight patter to the newAuthToken method.

    2) Modify the logout method so that it looks up what other tokens the User has out and also invalidates them. It would be ideal to add this method with a different signature so that the User may have the option of logging out a single token or logging out all authTokens. This would make the system the most flexible