Smart City Authentication Service Design Document

Date: 11/06/2020
Author: Mariam Gogia
Reviewer(s): Mildred Fakoya, Pawel Nawrocki

Table of Contents

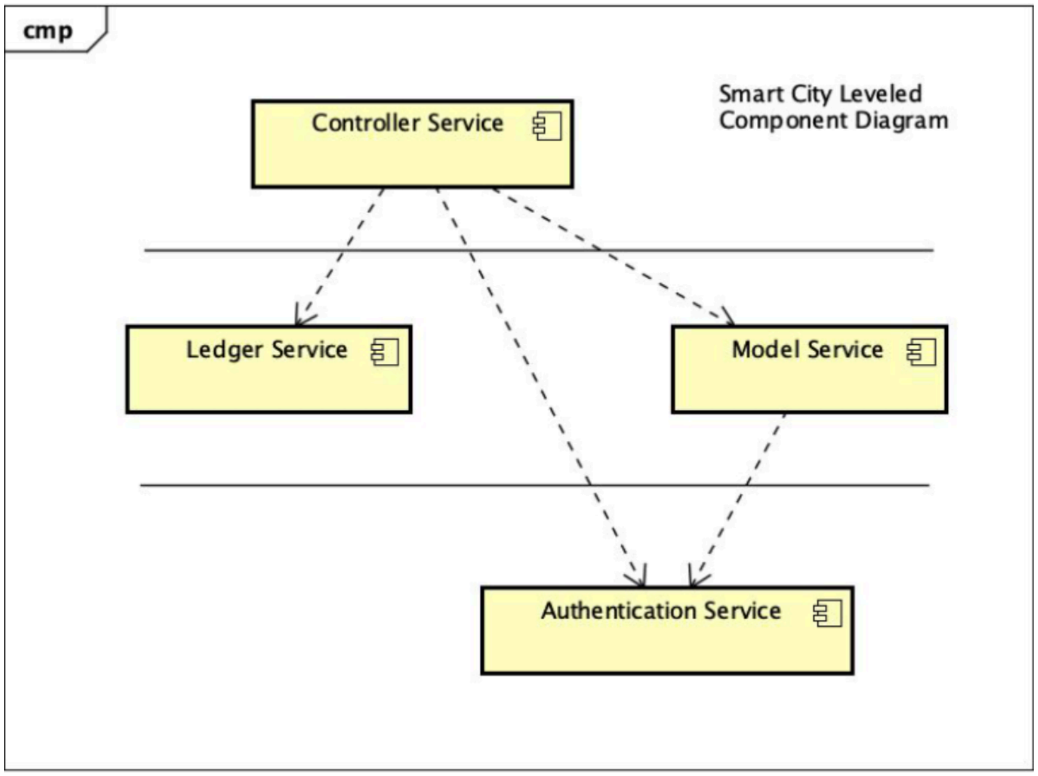
Introduction

This document defines the design for the Smart City Authentication Service

Overview

The authentication service represents one of the 4 modules of the Smart City System. This module is providing security to the Smart City by allowing only authenticated persons with the right access to interact with the City and its objects. Authentication service ensures that only entities with the right permissions and appropriate roles can add objects to the city, utilize and control IoT devices. Authentication service first identifies the user through biometrics, then asserts whether the user has appropriate permission to perform certain actions, after which it allows or denies access to that user.

Authentication is accessed by Model and Controller Services.



Requirements

This section provides a summary of the requirements for the Authentication Service.

The Authentication module should support the following:

- Control access to the Model Service
- Given the passed in auth_token, validate the associated user's permission to invoke requested method
- For every inhabitant, create a corresponding user within the Authentication Service
- Ensure that users within the Authentication Service are stored by their biometrics
- Public administrators should be assigned a Resource Role within the Authentication Service that binds the administrator with the city and appropriate role
- Update the Model Service accordingly to support Authentication service properly
- Update the Controller service so that it can request an auth_token from Authentication service to access Model Service API
- For voice commands coming from residents, Controller must look up auth_token given the resident's biometrics and use that auth_token when calling the Model Service methods.
- Authorize or deny request given the role/permission a certain resident has
- Update the command processor to support the Authentication Service Commands, including login command which returns the auth_token.

Software Design Requirements

- Apply **Visitor Pattern** to:

  I. Traverse through all Authentication Service objects and provide inventory of all Users, Resources, Access, Roles, and Permissions.'

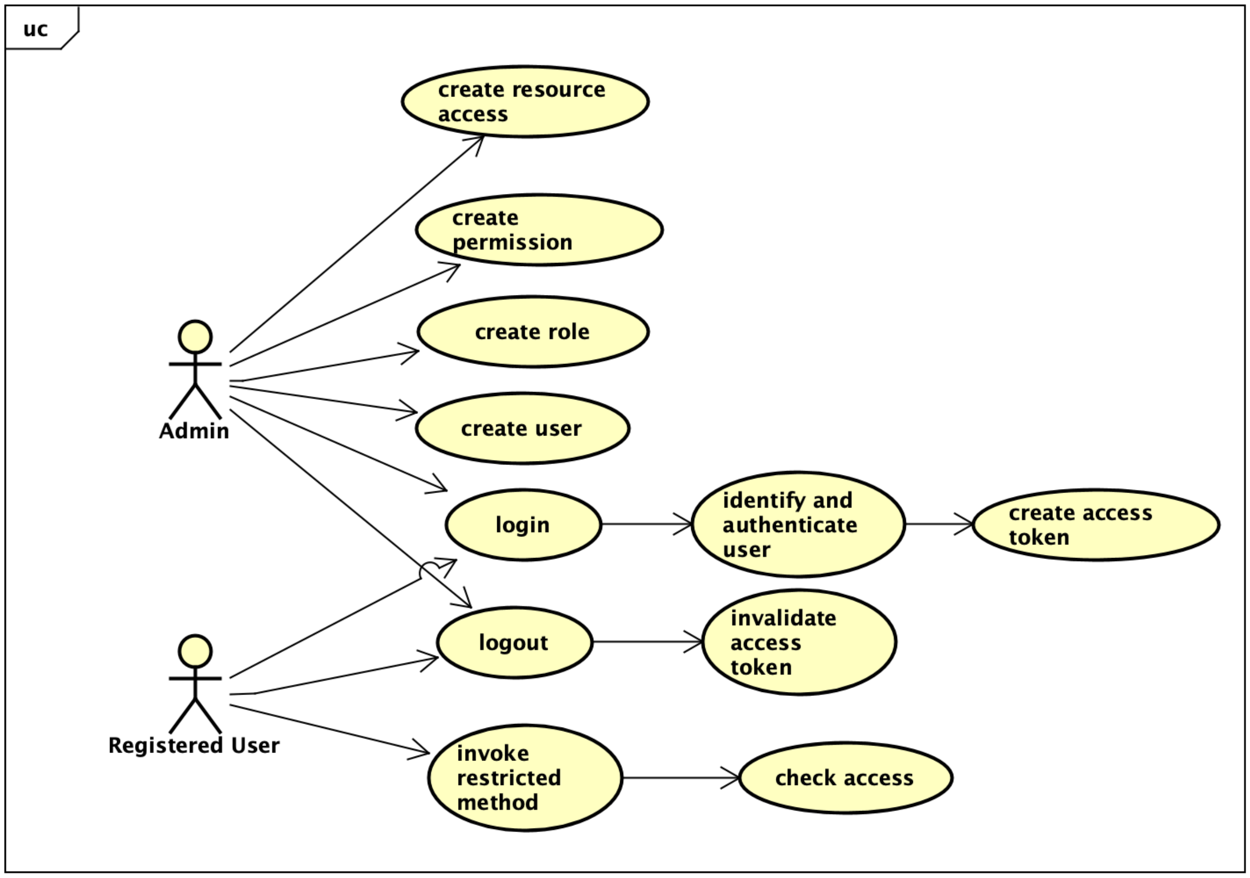
  II. Provide checking for access

- **Singleton Pattern** to ensure there is only one instance of the Authentication Service implementation accessed throughout the system.

- **Composite Pattern** to manage the whole part relation of Roles, Resources, Accesses, and Permissions.

Use Cases

The following diagram describes the use cases supported by the Authentication Service.



Actors:

Admin:

Responsible for managing Authentication service objects (Roles, Permissions, Users, Resources). It provisions cities and has full access to all Smart City System resources.

Registered User:

Users interact with the city using their biometrics or login credentials. Authentication service identifies authenticated users and approves or denies their request based on their permission and access.

Use Cases:

Create resource access:

Resource is a physical or logical entity, e.g. IoT. Admin can assign access to certain resources to certain users. Assigning a city administrator to specific city, is an example of creating resource access.

Create permission:

Permission represent the authorization required to do access resources within the city. An example of permission is ‘control_robot’ which allows the permission holder to control robots in the city. One user can be assigned zero or multiple permissions.

Create role:

Roles are composites of Permissions. Roles represent the group of permissions that user with particular role will get by default. It provides a way to logically group permissions and roles.

Create user:

For each user of smart city there is a mirroring user in Authentication service. Users have ids and a set of credentials, which may be their biometrics or username/password. Credentials are hashed for security. Users can be associated with 0 or more entitlements.

Login:

In order to interact in the city, every user must log in, through with they will obtain auth_token that will allow them to interact with he city.

- Identify and authenticate user:

  User logs in using his/her credentials (username, password) or biometrics (voice print, faceprint). Authentication Service verifies user’s existence and confirms that the hash of the password provided matches the known hashed password.
  If user logs in using biometrics, no hashing required.
- Create access token:

  If the user logs in successfully, he/she is issued an auth_token which allows him/her to interact with objects in the city as auth_token will be linked to the information about their Roles, Permissions, etc.

Logout:

User logs out from the system.

- Invalidate access token:

Upon log out assigned auth_token becomes invalid. Using invalidated token results in InvalidAuthTokenException.

Invoke restricted method:

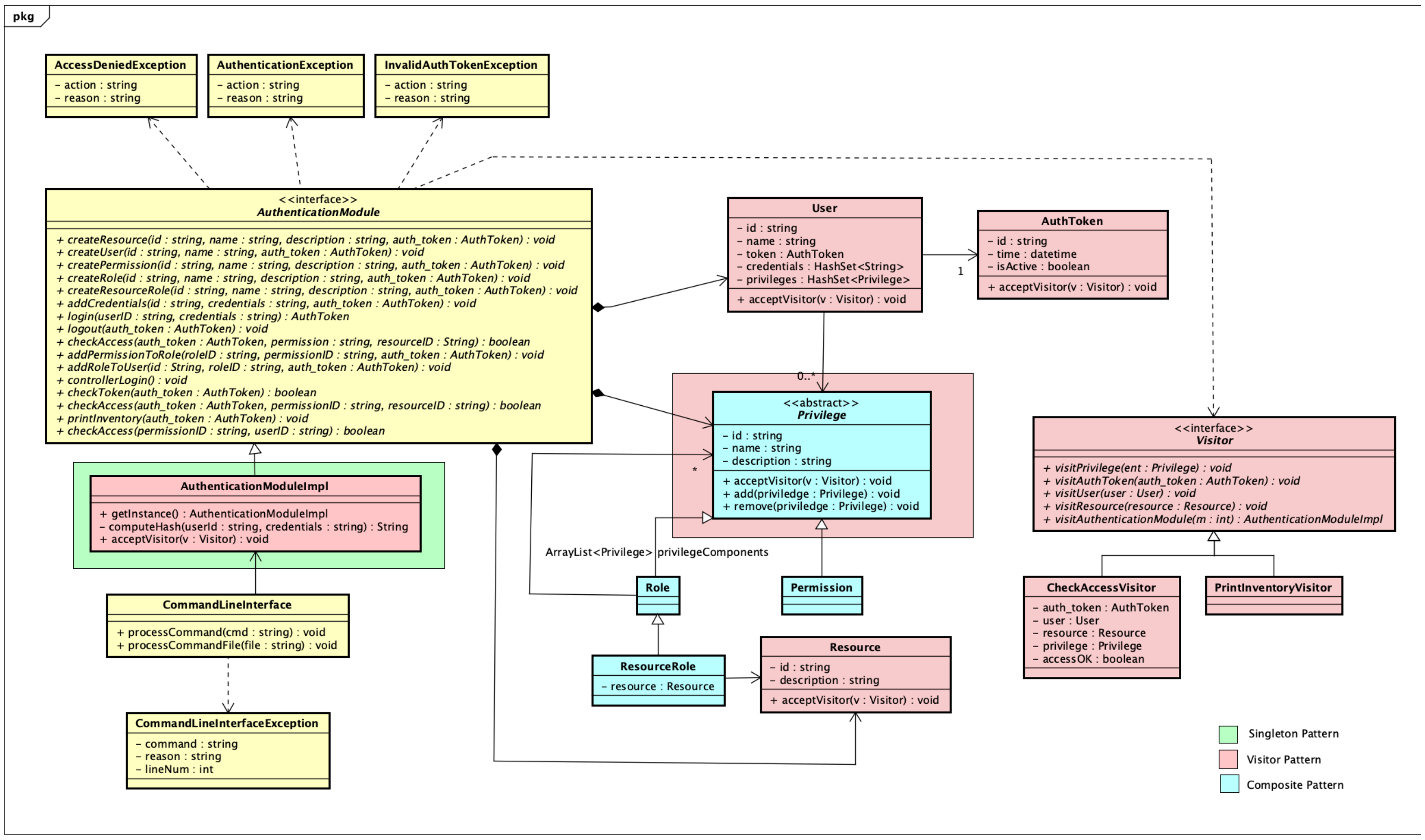
User uses the auth_token assigned at login to invoke methods int he Smart City. Each method validates that the token is not empty or null.

- Check Access:

  Smart City passes the auth_token to Authentication service with the required permission to invoke the method. Authentication service must check if the owner of auth_token has permission for the method he/she is trying to invoke, as well as check whether the auth_token is active - not expired.

Implementation

Class Diagram


pkg
AccessDeniedException
– action : string
– reason : string
AuthenticationException
– action : string
– reason : string
InvalidAuthTokenException
– action : string
– reason : string
<<interface>>
AuthenticationModule
+ createResource(id : string, name : string, description : string, auth_token : AuthToken) : void
+ createUser(id : string, name : string, auth_token : AuthToken) : void
+ createPermission(id : string, name : string, description : string, auth_token : AuthToken) : void
+ createRole(id : string, name : string, description : string, auth_token : AuthToken) : void
+ createResourceRole(id : string, name : string, description : string, auth_token : AuthToken) : void
+ addCredentials(id : string, credentials : string, auth_token : AuthToken) : void
+ login(userID : string, credentials : string) : AuthToken
+ logout(auth_token : AuthToken) : void
+ checkAccess(auth_token : AuthToken, permission : string, resourceID : String) : boolean
+ addPermissionToRole(roleID : string, permissionID : string, auth_token : AuthToken) : void
+ addRoleToUser(id : String, roleID : string, auth_token : AuthToken) : void
+ controllerLogin() : void
+ checkToken(auth_token : AuthToken) : boolean
+ checkAccess(auth_token : AuthToken, permissionID : string, resourceID : string) : boolean
+ printInventory(auth_token : AuthToken) : void
+ checkAccess(permissionID : string, userID : string) : boolean
AuthenticationModuleImpl
+ getInstance() : AuthenticationModuleImpl
– computeHash(userId : string, credentials : string) : String
+ acceptVisitor(v : Visitor) : void
CommandLineInterface
+ processCommand(cmd : string) : void
+ processCommandFile(file : string) : void
CommandLineInterfaceException
– command : string
– reason : string
– lineNum : int
User
– id : string
– name : string
– token : AuthToken
– credentials : HashSet<String>
– privileges : HashSet<Privilege>
+ acceptVisitor(v : Visitor) : void
1
AuthToken
– id : string
– time : datetime
– isActive : boolean
+ acceptVisitor(v : Visitor) : void
0..*
<<abstract>>
Privilege
– id : string
– name : string
– description : string
+ acceptVisitor(v : Visitor) : void
+ add(priviledge : Privilege) : void
+ remove(priviledge : Privilege) : void
*
ArrayList<Privilege> privilegeComponents
Role
Permission
ResourceRole
– resource : Resource
Resource
– id : string
– description : string
+ acceptVisitor(v : Visitor) : void
<<interface>>
Visitor
+ visitPrivilege(ent : Privilege) : void
+ visitAuthToken(auth_token : AuthToken) : void
+ visitUser(user : User) : void
+ visitResource(resource : Resource) : void
+ visitAuthenticationModule(m : int) : AuthenticationModuleImpl
CheckAccessVisitor
– auth_token : AuthToken
– user : User
– resource : Resource
– privilege : Privilege
– accessOK : boolean
PrintInventoryVisitor
🟩 Singleton Pattern
🟥 Visitor Pattern
🟦 Composite Pattern

***Not shows for readability purposes:*

**Classes with acceptVisitor methods implement from Visitable interface*

**Visitor Interface is associated with all the objects in the system.*

Class Dictionary

Model Service Modifications

Every function in the model service calls the checkAccess function in the Authentication service, which checks if the logged in user is allowed to perform certain actions. Added a helper method checkAccess, which is called by each method in the API.

*** Observe that the update method now takes in auth token to pass it to concrete classes (commands) and assess whether the logged-in user has permission to invoke a certain rule and actions that that rule entails.*

Method Name	***Signature***	***Description***
checkAccess	(token:AuthToken, permissionID:string, resourceID:String):boolean	Calls checkAccess function in authentication module which checks whether user has permission to invoke the method.

Controller Service Modifications

Execute method of each command class takes in auth token. Each execute method calls checkAccess on authentication module to determine whether the logged in user has permission to invoke the rule.

Interfaces & Abstract Classes

- AuthenticationModule - provides all the functions to be implemented by Authentication Module Implementation
- Visitor Interface - provides the interface for Visitor classes, namely - CheckAccessVisitor and PrintInventoryVisitor. It provides 5 methods to visit 5 different objects in the module.
- Visitable Interface - provides acceptVisitor method for 5 visitable classes.
- Privilege Abstract Class - composable class of the composite pattern, it provides structure for privileges like Role, ResourceRole and Permission.

Dependencies, Associations & Compositions

Dependencies:

- Visitor interface, and therefore it's implementing classes depend on Authentication Module. Visitor cannot carry out its work without Authentication Module.
- All three types of exceptions depend on Authentication module.

Associations:

- User and tokens have one-to-one relationship - 1 user can only have 1 token at a time.
- User and Privileges have zero-to-many relationships- 1 user can have zero or many privileges.
- Role and Privilege have many-to-many relationship.
- CommandLineInterface calls methods in AuthenticationModuleImpl.
- ResourceRole is associated with Resource because it uses Resource as one of its properties.
- Visitor classes have associations with all the objects they are visiting (not shown for readability purposes).

Compositions:

- Authentication module is has a composite relationship with most of its objects as objects like users, privileges, resource, and tokens cannot exist without Authentication module - it's the one defining them, adding them to the system and updating them if necessary.

Classes

***AuthenticationImpl** implements Authentication interface and provides the API for authentication module. It's 'create' methods fulfill the project requirement of defining objects (resources, users, roles, permissions) in the module. The class applies a singleton pattern by providing a single instance of itself throughout the whole system.*

Property Name	***Type***	***Description***
userCredentials	Map <String, String>	Map uses unique user ids as its key and stores the hash of the corresponding user credentials as its value. It is used to verify user credentials upon login.
users	Map <String, User>	Map storing all the users by their ids.
resources	Map <String, Resource>	Map storing all the resources by their ids.

privileges	Map <String, Privilege>	Map storing all the privileges (roles, permissions, resource roles) by their ids.
currentUser	User	Represents currently logged in user

Method Name	***Signature***	***Description***
getInstance	():AuthenticationImpl	Returns the instance of the module (part of the singleton pattern)
createResource	(id:String, description:String, token:AuthToken):void	Checks the validity and the access of the passed in token, if OK, adds the resource to the module, otherwise it throws the appropriate exception
createUser	(id:String, name:String, token:AuthToken):void	Checks the validity and the access of the passed in token, if OK, adds the user to the module, otherwise it throws the appropriate exception
createPermission	(id:String, name:String, description:String token:AuthToken):void	Checks the validity and the access of the passed in token, if OK, adds the permission to the module, otherwise it throws the appropriate exception
createRole	(id:String, name:String, description:String token:AuthToken):void	Checks the validity and the access of the passed in token, if OK, adds the role to the module, otherwise it throws the appropriate exception
createResourceRole	(id:String, name:String, description:String token:AuthToken):void	Checks the validity and the access of the passed in token, checks if such resource and role exists in the system - if OK, adds the resource role to the module, otherwise, it throws the appropriate exception

addCredentials	(id:String, credentials:String, token:AuthToken):void	Checks the validity and the access of the passed in token, checks if such user exists, if OK - hashes the passed in credentials and adds it to the user - as well as to the map of userCredentials, otherwise, it throws the appropriate exception
addPermissionToRole	(roleID:String, permissionID:String, token:AuthToken):void	Checks the validity and the access of the passed in token, checks if such role and permission exist, if OK - adds permission to the role, otherwise, it throws the appropriate exception
addRoletoUser	(userID:String, permissionID:String, token:AuthToken):void	Checks the validity and the access of the passed in token, checks if such user and role exist, if OK - adds role to the user, otherwise, it throws the appropriate exception
login	(userID:String, credentials:String):AuthToken	1. If userID is "**super_admin**" and its credential is "**initializeservice**" : initialize user with admin privileges, create permissions to create, update, delete privileges, users, and resources; to charge ledger accounts and simulate events and add it to 'admin_role', assign the admin role to admin user. Assign the token and return it. 2. Otherwise, check if the hashes of user credentials match, if so - assign and return token. 3. Throw the appropriate exception
logout	(token:AuthToken):void	Sets token's isActive field to false
checkToken	(token:AuthToken):boolean	Checks whether token is expired or not (active for > 60 minutes), if token is OK, returns true, if its expired return false.

checkAccess	(token:AuthToken, permissionID:String, resourceID:String):boolean	Checks the validity of the token using a visitor pattern, It visits different visitable classes to determine whether the token owner has access to the function or not. Its resourceID field must be empty if the check for ResourceRole is not intended.
checkAccess	(permissionID:String, userID:String)	Checks if the user has authorization to do certain things in the city - e.g. ride a bus.
computeHash	(id:String, credentials:String):String	Using the userID and credentials and SHA-256 algorithm, it computes the hash.
acceptVisitor	(visitor:Visitor):void	(part of the visitor pattern) checkAccess function passes in the visitor object to acceptVisitor function where acceptVisitor calls visitAuthenticationModule on it.

***Authtoken** - class represents the authentication token for the module. It is a part of visitor pattern and implements Visitable interface. Class satisfies the requirement of having auth tokens in the system. It is issued to every user upon login.*

Property Name	***Type***	***Description***
id	String	Token unique ID, consists of userID and the time it was created.
isActive	boolean	Indicates whether the token is active or expired.
lastUsed	LocalDateTime	The date and time token was last used.

Method Name	***Signature***	***Description***
acceptVisitor	(visitor:Visitor):void	(part of the visitor pattern) the class calls visitAuthToken function on passed in visitor and passes in the instance of the token (this).

***Privilege** - the abstract class represents the privileges(entitlements) in the module and as Privilege and its extending classes (Role and Permission) are part of the visitor pattern - the Privilege class implements the Visitable Interface. Privilege class provides the structure to entitlement objects as per requirements. Privilege class is part of the composite pattern it represents the composable component of the pattern.*

Property Name	***Type***	***Description***
id	String	Privilege's unique ID
name	String	The name of the privilege
description	String	The description of the privilege

Method Name	***Signature***	***Description***
abstract add	(privilege:Privilege):void	Method to add a privilege
abstract remove	(privilege:Privilege):void	Method to remove a privilege
acceptVisitor	(visitor:Visitor):void	(part of the visitor pattern) the class calls visitPrivilege function on passed in visitor and passes in the given instance of the privilege (this).

***Role -** the class represents roles in the module, it is a part of visitor and composite patterns. It represents the composite class of the composite pattern.*

Property Name	***Type***	***Description***
privilegeComponents	ArrayList<Privilege>	Stores all the privileges of the role

Method Name	***Signature***	***Description***
add	(privilege:Privilege):void	Adds the privilege to the privilegeComponents list
remove	(privilege:Privilege):void	Removes the privilege to the privilegeComponents list
getPrivilegeComponents	():ArrayList	Getter for privilegeComponents list

getPrivilegeComponents	(i:int):Privilege	Gets privilegeComponent by its index in the list.

Permission *- the class represents permissions in the module, it is a part of composite and visitor patterns, it extends the Privilege class and implements Visitable interface. It represents the leaf of the composite pattern, for this reason, it does no provide implementations of add and remove methods. It does not have extra methods or properties. It uses super constructor.*

ResourceRole - *the class represents resource roles in the system and is part of composite and visitor patterns, it extends Role which itself extends Privilege.*

Property Name	***Type***	***Description***
resource	Resource	Resource to be managed by a certain role

User - *the class represents the user in the authentication module, it is a part of visitor pattern and implements the Visitable interface. It helps in fulfilling the requirement of providing users for every inhabitant in the system.*

Property Name	***Type***	***Description***
id	String	User’s unique ID
name	String	User's name
token	AuthToken	User’s token
credentials	HashSet<String>	Set of user’s credentials (hashed)
privileges	HashSet<Privilege>	Set of user’s privileges

Method Name	***Signature***	***Description***
addCredential	(credential:String):void	Method to add a user credential to its credentials
addPrivilege	(privilege:Privilege):void	Method to add privilege to user’s privileges

acceptVisitor	(visitor:Visitor):void	(part of the visitor pattern) the class calls visitUser function on passed in visitor and passes in the given instance of the user (this).

***Resource -** the class represents the resource in the authentication module, it is a part of visitor pattern and implements the Visitable interface. It helps in fulfilling the requirement of defining resources in the system.*

Property Name	***Type***	***Description***
id	String	resource ID
description	String	resource description

Method Name	***Signature***	***Description***
acceptVisitor	(visitor:Visitor):void	(part of the visitor pattern) the class calls visitResource function on passed in visitor and passes in the given instance of the resource (this).

Visitor Classes

***CheckAccessVisitor -** Visitor class to check whether the certain token holder has access to requested functions in the system. It fulfills the requirement of providing access control for different modules of the system.*

Property Name	***Type***	***Description***
user	User	token owner user
resource	resource	access to which to be checked
token	AuthToken	token accesses of which to be checked
permission	Privilege	permission that user should have to access the function
accessOK	boolean	Indicates whether the user has access or not

Method Name	***Signature***	***Description***
visitAuthenticationModule	(authenticationImp:AuthenticationImpl):void	If the token belongs to super admin, call acceptVisitor(this) on super admin, otherwise, determine which user is currently logged in and call acceptVisitor(this) on him/her.
visitUser	(user:User):void	Set the user to passed in user, call acceptVisitor(this) on token.
visitAuthToken	(authToken:AuthToken):void	Check if user's token and token match, if so, iterate in user's privileges and call acceptVisitor(this) on each privilege user has.
visitPrivilege	(privilege:Privilege):void	If permission id and passed in privilege's id equal, set accessOK to true, if passed in privilege is of a Role class - iterate in role's privilegeComponent list - if privilege id from privilegeComponents list matches the permission id of the visitor - set accessOK to true. If privilege class is of ResourceRole, get the resource of the resource role and call acceptVisitor(this) to the resource.
visitResource	(resource:Resource):void	If resource id of the class property and the passed in resource id equal - set accessOK to true.

PrintInventoryVisitor - *Visitor class to print inventory of all users, resources, roles, and permissions.*

Method Name	***Signature***	***Description***
visitAuthenticationModule	(authenticationImp:AuthenticationImpl):void	Prints the name of the module, iterates through the list of all users and calls acceptVisitor(this_ on each

visitUser	(user:User):void	Print the details of the of user, if the user token is not null call acceptVisitor(this) on token. Iterate through users privileges and call acceptVisitor(this) on each.
visitAuthToken	(authToken:AuthToken):void	Print the details of the token
visitPrivilege	(privilege:Privilege):void	Print the details of the privilege. If privilege is of a Role class, iterate through its privilegeComponents and call acceptVisitor(this) on each permission. If the privilege is of ResourceRole iterate through its privilegeComponents and call acceptVisitor(this) on each.
visitResource	(resource:Resource):void	Print the details of the resource

Exception Classes

AuthenticationException

Property Name	***Type***	***Description***
action	String	Performed action (e.g. createUser)
reason	String	Reason that caused the exception (user already exists)

AccessDeniedException

Property Name	***Type***	***Description***
action	String	Performed action (e.g. checkAccess - createUser)
reason	String	Reason that caused the exception (token owner not authorized for this action)

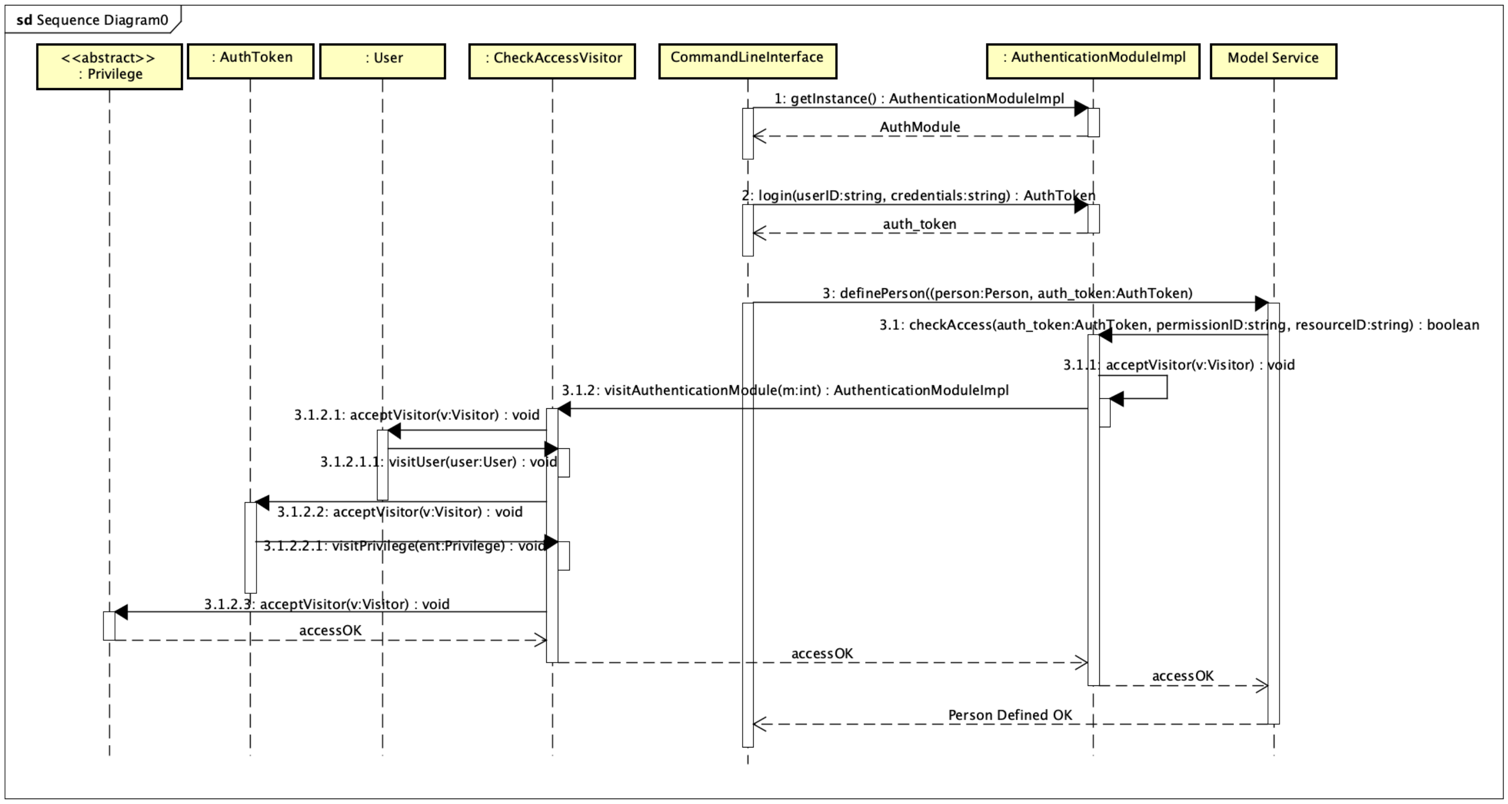
InvalidTokenException

Property Name	***Type***	***Description***
action	String	Performed action (e.g. checkAccess - createUser)

reason	String	Reason that caused the exception (token expired)

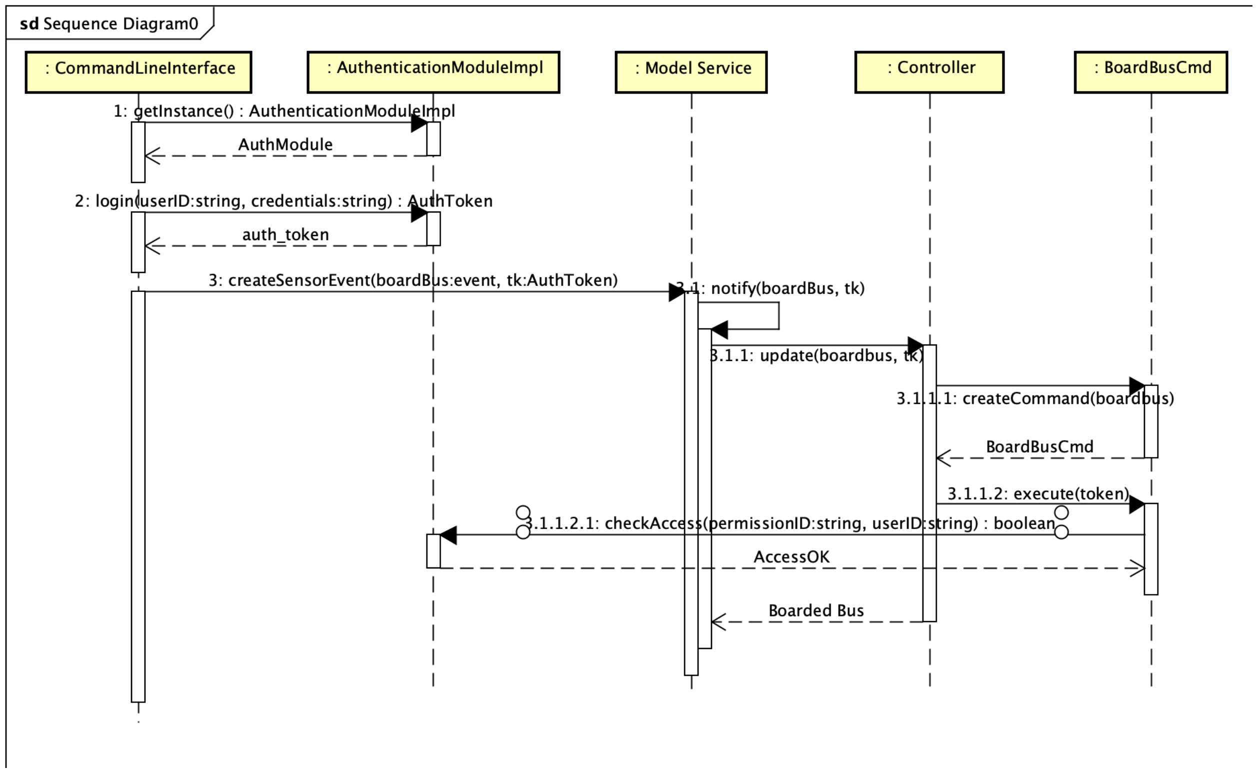
Implementation Details

Sequence Diagram - showing definePerson method and how the access is being checked using the visitor pattern.


sd Sequence Diagram0
<<abstract>> : Privilege
: AuthToken
: User
: CheckAccessVisitor
CommandLineInterface
: AuthenticationModuleImpl
Model Service
1: getInstance() : AuthenticationModuleImpl
AuthModule
2: login(userID:string, credentials:string) : AuthToken
auth_token
3: definePerson((person:Person, auth_token:AuthToken)
3.1: checkAccess(auth_token:AuthToken, permissionID:string, resourceID:string) : boolean
3.1.1: acceptVisitor(v:Visitor) : void
3.1.2: visitAuthenticationModule(m:int) : AuthenticationModuleImpl
3.1.2.1: acceptVisitor(v:Visitor) : void
3.1.2.1.1: visitUser(user:User) : void
3.1.2.2: acceptVisitor(v:Visitor) : void
3.1.2.2.1: visitPrivilege(ent:Privilege) : void
3.1.2.3: acceptVisitor(v:Visitor) : void
accessOK
accessOK
accessOK
Person Defined OK

First, the CommandLineInterface gets the singleton of AuthenticationModuleImpl, then the user logs in, after successful log in (which is assumed here), user gets auth_token assigned. This diagram intends to test 'definePerson' and this method requires public_admin_role. Let's assume that the logged in person is a public admin. Admin fires 'definePerson' method passing the token he just received. Model service's define person function calls checkAccess function in AuthenticationModuleImpl. CheckAccess assembles the CheckAccessVisitor visitor object and calls acceptVisitor on it. Visitor pattern iteration is demonstrated on the left part of the diagram. Assuming that the admin has access to define the person, it returns accessOK and the person is defined.

Sequence Diagram Demonstrating another checkAccess method. This check access method is used to determine whether certain users have authorization to utilize resources in the city. It is different from the one checking the authorizations of currently logged in user.


sd Sequence Diagram0
: CommandLineInterface
: AuthenticationModuleImpl
: Model Service
: Controller
: BoardBusCmd
1: getInstance() : AuthenticationModuleImpl
AuthModule
2: login(userID:string, credentials:string) : AuthToken
auth_token
3: createSensorEvent(boardBus:event, tk:AuthToken)
3.1: notify(boardBus, tk)
3.1.1: update(boardbus, tk)
3.1.1.1: createCommand(boardbus)
BoardBusCmd
3.1.1.2: execute(token)
3.1.1.2.1: checkAccess(permissionID:string, userID:string) : boolean
AccessOK
Boarded Bus

After getting the instance of the AuthenticationModule and logging in, board-bus event is simulated. The model service calls its notify method, which calls the update method - passing in the event (boardBus) and the token. Controller service, given the information - creates the appropriate command (BoardBusCmd) and calls execute method on it. Execute calls checkAccess where permission and userIDs are passed in and checks whether the user requesting to board bus has authorization to ride a bus or not.

Other Implementation Details and Comments

Super user credentials are username: **super_admin** password: **initializeservice**
Upon super user log in - super user permissions and the role are internally initialized.

These permissions include: ‘auth_user_admin,’ ‘auth_role_entitlement_admin,’ etc. Upon super admin log in, command line interface calls ‘controller_login() and acquires controller_auth_token.

Permissions: “scms_charge_users” and “scms_simulate_event” were added to super admin privileges. They are used to check whether the logged in user can charge users ledger accounts and whether the logged in user has authorization to simulate sensor event.

Token expiry time is set to 60 minutes.
Controller and super_admin have same privileges and controller uses super_admin’s tokens. This design may be subject to change.

Code compiles using the following (from Piazza):

```
javac cscie97/smartcity/model/*.java cscie97/smartcity/controller/
*.java cscie97/smartcity/authentication/*.java com/cscie97/ledger/
*.java cscie97/smartcity/test/*.java
```

Runs:

```
java -cp . cscie97.smartcity.test.TestDriver smart_city.script
```

Exception Handling

Authentication module adds 3 types of exceptions to the system.

- AuthenticationException - invoked when there are errors in defining objects or enquiring the module
- InvalidAuthTokenException - invoked when token is no longer valid (expired)
- AccessDenidedException - invoked when logged in user has no access to the method he/she is requesting

Testing

In addition to the given script, another tester file - smart_city_2.script is provided. The file is documented itself and gives descriptions of expected values before each command.

The file test for:

- Not allowing creation of duplicate privileges
- Currently logged in user's attempt to access the method he/she has no authorization for
- Attempt to define objects with invalid token
- Log in with invalid credentials
- Printing inventory

Script Commands:
Script commands have been modified and are as follows:

```
log in <userId> <credentials>
define permission <permission_id> <permission_name> <permission_description>
define role <role_id> <role_name> <role_description>
add_permission to_role <role_id> <permission_id>
create user <user_id> <user_name>
add user_credential <user_id> <biometric|password> <value>
add_role to_user <user_id> <role>
define resource <resource_id> <description>
create resource_role <resource_role_name> <role> <resource>
log out
```

Risks & Limitations

Integration of Controller service with Authentication service needs to be improved. In addition, currently, the system only supports one token at a time. It does not support multiple logged in users. Users need to be manually defined in both services via script. Better approach would be dynamic creation of users from model service to authentication service.