Smart City Authentication Service Requirements

Brett Bloethner

Introduction

This document provides system architecture details for the Smart City Software System's Smart City Authentication Service. In this document you will find use case diagrams, class diagrams, sequence diagrams and implementation details that will help outline and explain the Smart City Authentication Service design.

Table of Contents

1. Overview
2. Requirements
3. Use Cases & Use Case Diagram
4. Class Diagram
5. Class Dictionary
6. Implementation Details
7. Exception Handling
8. Testing
9. Risks
10. System Changes

Overview

The Smart City Authentication Service is responsible for securing the Smart City against unauthorized access. It does that by working with the Model and Controller services to manage users and authentication tokens. The Model Service then uses the authentication tokens to ensure the requester has the correct privileges for the action they're trying to perform. There are three design patterns that are heavily incorporated into the Authentication Service and allow it to efficiently manage roles, permissions and users. These three design patterns are the visitor pattern, then singleton pattern and the composite pattern.

Requirements

This section describes the features required of the Authentication Service. In order to effectively support the other components of the Smart City System, the Smart City Authentication Service must…

Create resources…

- Resources should be able to be made with a unique identifier and description. Resources represent a physical entity in the smart city such as an IoT devices. Once resources are made, permissions can be given to roles in order to access the resource.

Create permissions…

- Permissions exist to allow a user access to a resources or function in the Smart City system. A user may have zero permissions but they're not limited in how many permissions they could have. Roles can also include permissions and a user can be associated with roles, permissions or both. Each permission includes a unique ID a name and a description.

Create roles…

- Roles provide an easy and logical way to group permissions and are a composite of permissions. They also include a unique ID, a name and a description. If a user is associated with a role, the user then has the privileges provided by each permission in that role and in sub roles of that role.

Create users…

- The system must support the creation of users. Users represent a person in the smart city system and for each person in the model service, a user would also exist representing that same person. Each user includes a unique ID, a name and at least one credential which could be either username/password, voiceprint, or faceprint. Users can be associated with zero or many roles and permissions. For security, the credentials used to authenticate a user are stored using a hashing algorithm.

Allow login and the issuing of authentication tokens…

- The system should be able to authenticate a user via the user credentials. An auth token will then be returned after which the client can use the auth token to execute methods in the Model Service on the users' behalf. The auth token, through the associated user, is bound to the permissions and roles that the Auth Service checks to verify the user can access certain features. The returned auth token should include a unique ID, expiration time (after which the token will stop working), state (active or expired), Entitlements and an association to the User to whom the Token belongs. If the authentication attempt is unsuccessful, an Authentication Exception must be thrown.
- Validation of the credentials is done by first making sure the user exists and then proceeding to compare the hash of the input credential with the stored credential.
- Biometric authentication should be simulated via strings ("voice-print='voiceprint-jane'") and a matching pattern similar to what's done with the hashes.

Allow log out and the invalidation of authentication tokens…

- Logging a user out must mark the auth token as invalid, that way the token can no longer be used.

Allow clients to check the access status of a user…

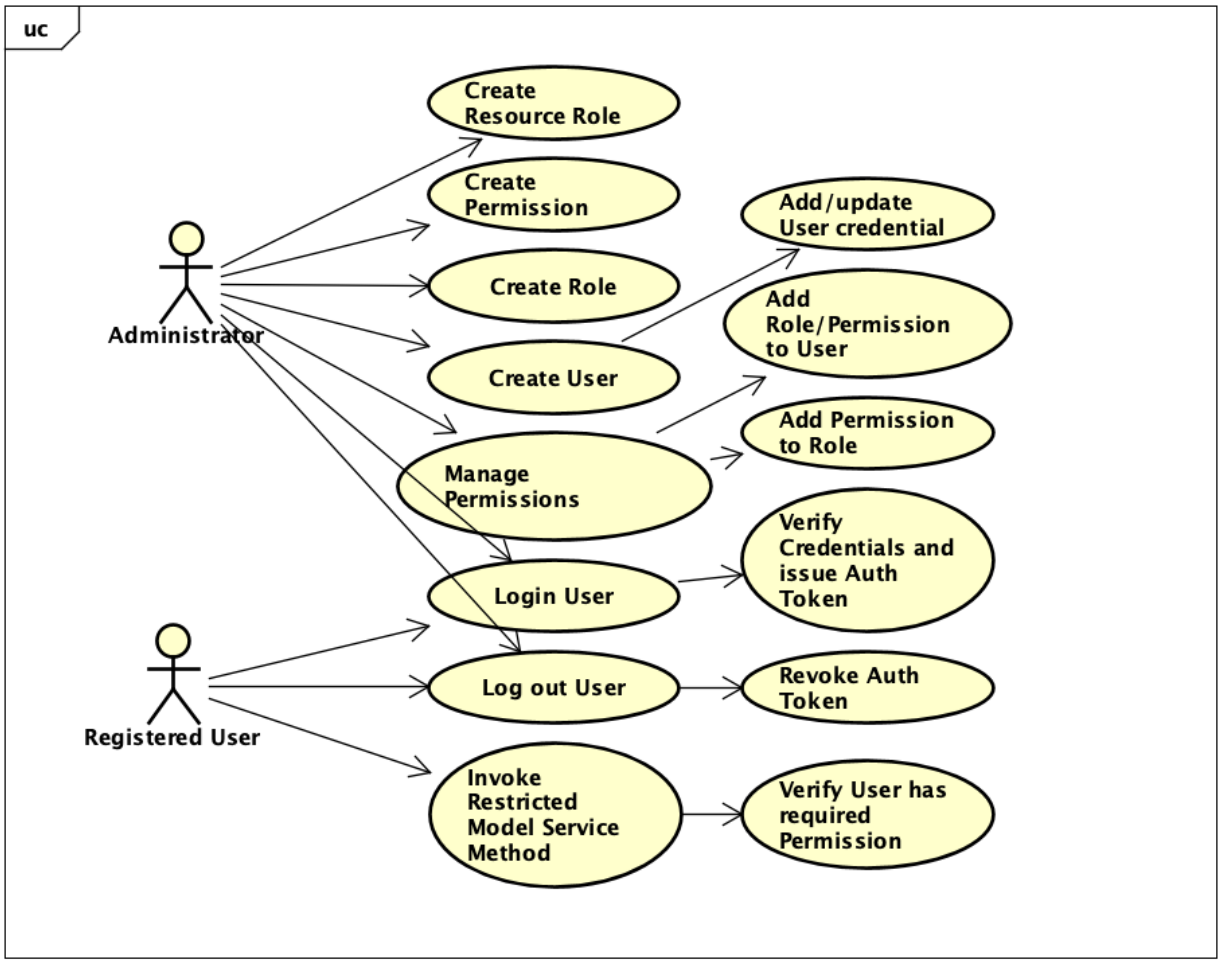
- Checking access allow the model service to verify that the client has the appropriate privileges to execute the method they're trying to execute. Methods in the Model Service must first verify that the auth token is passed into the method. Next, they must send the auth token along with the permission ID associated with that Model Service method. Once the Auth Service verifies that the auth token is valid and not expired, the Auth Service will check to verify that the user associated with the auth token does in fact have the correct Permission to execute the method on the Model Service.
- The check access pattern must be implemented on every method in the Model Service.

Use Cases

Actors

- **Administrator:** The administrator is responsible for managing the Smart City Authentication Service. To do this, they must have the ability to create Resource Roles for things like cities and IoT devices, have the ability to create Permissions, have the ability create Roles and Users as well as the ability to associate those Users to Permissions and Roles. In order to effectively manage the Smart City Authentication Service, the administrator must have full access to Smart City Resources and Users in the system.
- **Registered User:** The registered user is the actor who authenticates against the Smart City Authentication System through other services like the Controller and Model services. They have the ability to login via username and password or via biometrics. Additionally, once a registered user is logged in, an auth token will be issued which lets the registered user prove their identity to the Model Service for actions they request the Smart City System execute on their behalf.

Below is the use case diagram which outlines what the expected actors are and what abilities they'll have in the application.



Create Resource Role

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that a Resource Role with that ID doesn't already exist
- Verify the Resource exists by ID in the Model Service

- Create an associate to each other the Resource Role and Resource

Create Permission

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that a Permission with that ID doesn't already exist
- Create the new Permission with the correct ID, name and description
- Store the new Permission in the Authentication Service singleton
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything else above fails

Create Role

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that a Role with that ID doesn't already exits
- Create the new Role with the correct ID, name and description
- Store the new Role in the Authentication Service singleton
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything else above fails

Create User

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that the user ID provided exists for a User in the Model Service and get user
- Create the user in the Authentication Service
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything

Add Role to User

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that the user ID provided exists in the Authentication Service
- Verify that the role ID provided exists in the Authentication Service
- Add an association of the Role to the User
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything else above fails

Add Permission to Role

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that both the Permission and the Role exist
- Add an association of the Permission to the Role
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything else above fails

Add User Credential

- Verify that the auth token used in this action is valid against the correct Permission
- Verify that the user ID provided exists in the Authentication Service
- Create a new Credential object with the correct type and add it to the User
- Throw InvalidAccessTokenException if the auth token provided is invalid, AuthServiceException if anything else above fails

Login

- Verify that the user ID provided exists in the Authentication Service
- Verify that the hashes of the provided credential and the stored one match
- Create a new Authentication Token and associate it with the user
- Store the new Authentication Token in the Authentication Service singleton
- Return the auth token
- Throw AccessDeniedException if login credentials are invalid or user does not exist, AuthServiceException if anything else above fails

Logout

- Verify that the user ID provided exists in the Authentication Service

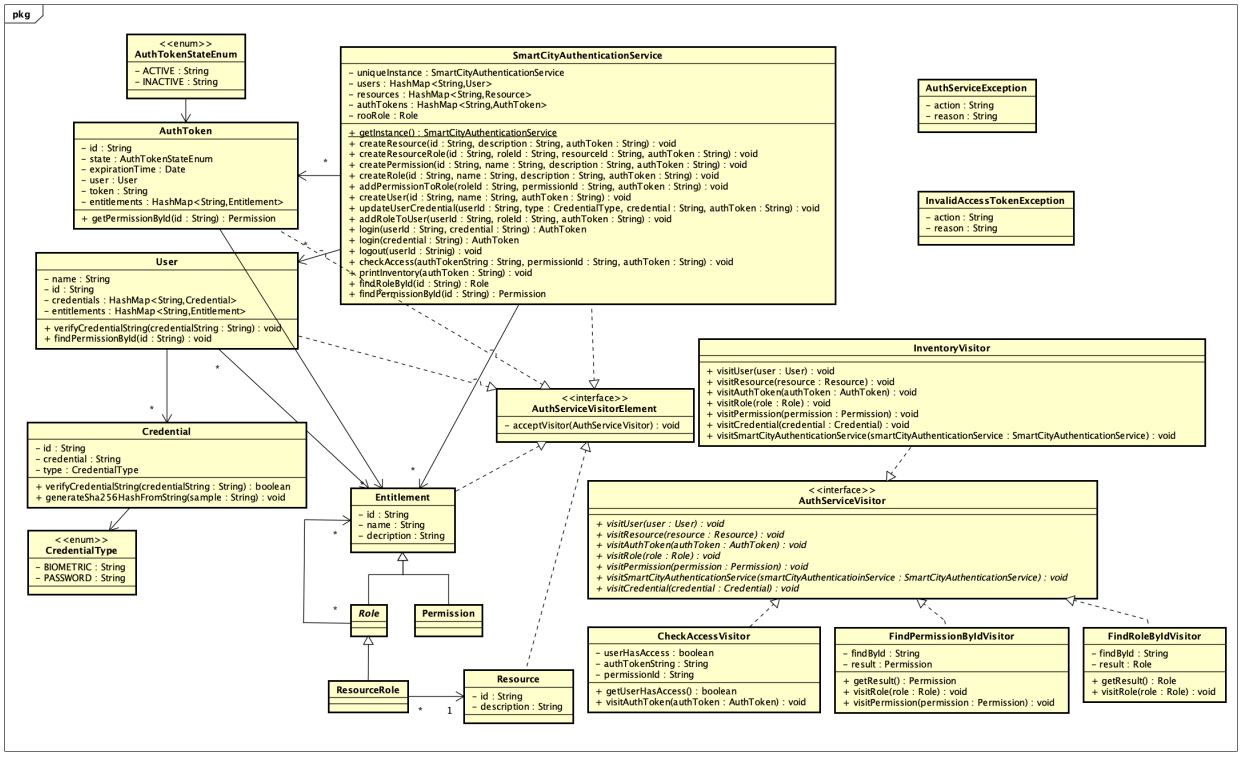
- Get the Users auth tokens from the Authentication Service
- If any auth tokens are active, then set them as inactive
- Throw AuthServiceException if anything above fails

Check Access

- Verify that the auth token provided matches one stored in the Authentication Service and that its active
- Get the AuthToken and iterate through the AuthTokens Entitlements
- If an Entitlement can be found with the provided permissionId then proceed error free
- Throw InvalidAccessTokenException if the auth token provided is invalid, AccessDeniedException if the user doesn't have access to the required resource or method (ie cant find an Entitlement)

Class Diagrams

The following class diagram defines the Smart City Authentication Service implementation classes contained within the package "com.cscie97.smartcity.authentication". Below is the class diagram outlining the design and relation of all the classes in the Smart City Authentication Service.



Class Dictionary

This section specifies the class dictionary for the Smart City Authentication Service. The classes should be defined within the package "com.cscie97.smartcity.authentication".

AuthToken *implements AuthServiceVisitorElement*

The AuthToken represents a token used by the system to validate against Permissions before a method is executed. AuthTokens have a Entitlement composition tree that reflects what Roles and Permissions they have access to, it is also associated with only one User.

Methods

Method Name	Signature	Description
getPermissionById	(id: String): Permission	Iterate through the Entitlements and use the FindPermissionByIdVisitor to traverse down each Entitlement tree and return the result that was got by the Visitor. This getPermissionById only gets a Permission by ID within the scope of the Permissions associated with the Auth Token.
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

Properties

Property Name	Type	Description
id	String	Unique Id of the Token
expirationTime	Date	Time of expiry for the Token
state	AuthTokenStateEnum	Active/Inactive state of the Token
user	User	User associated with the token
token	String	Token in its friendly plain text format

Associations

Association Name	Type	Description
entitlements	HashMap<String. Entitlement>	Map of Entitlements the Token has
user	User	User associated with the Token

CheckAccessVisitor *implements AuthServiceVisitor*

This visitor is sent through the Auth Token map where it visits each Auth Token and uses the FindPermissionById visitor (via authToken.getPermissionById()) to go through the AuthTokens Entitlements until it completes and can report whether or not it found a Permission with the ID permissionID provided.

Methods

Method Name	Signature	Description
getUserHasAccess	(): Boolean	get the result for after the visitor has completed
visitAuthToken	(authToken: AuthToken): void	Visit each Auth Token and if the auth token matches the one provided in the constructor and the auth token is not expired and the auth token is active and the auth token contains a Permission with the same ID as the one provided in the constructor, then set userHasAccess to true to

		show that the AutToken has access to the required Permission

Properties

Property Name	Type	Description
userHasAccess	Boolean	Stores results of whether or not token has access to the Permission ID
authTokenString	String	The Auth Token string in question
permissionId	String	The Permission ID string in question

Credential *implements AuthServiceVisitorElement*

This class is responsible for storing and verifying credentials. Users are associated with 0 or more Credentials and those Credentials are used to log users in. This class also hashes the credentials for security and does the validation for possible String matches to the stored hashed credential.

Methods

Method Name	Signature	Description
verifyCredentialString	(credentialString: String): Boolean	Verify a provided credential string against the stored hashed credential
generateSha256HashFromString	(sample: String): String	Utility method for creating a SHA256 hash, used from my Assignment 1
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

Properties

Property Name	Type	Description
id	String	An ID unique to all Credentials
credential	String	Hashed variant of the Credential
type	CredentialType	Type of Credential (BIOMETRIC or PASSWORD)

Entitlement *implements AuthServiceVisitorElement*

The Entitlement represents an action a User is entitled to and is the bases of the Role and Permission class which extend Entitlement. This class is not instantiated on its own, rather Permission and Role are used to extend it.

Methods

Method Name	Signature	Description
addEntitlement	(entitlement: Entitlement): void	Returns an unsupported exception, implemented in some subclasses
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	Not implemented, implemented in some subclasses

Properties

Property Name	Type	Description
id	String	ID unique to all Entitlements
name	String	Friendly name for the Entitlement
description	String	Friendly description for the Entitlement

FindPermissionByIdVisitor *implements AuthServiceVisitor*

This Visitor is used to iterate through the Entitlement composition tree until it finds a Permission that has the same ID provided to the Visitor.

Methods

Method Name	Signature	Description
getResult	():Permission	Get the result of what the Visitor finds or null
visitRole	(role: Role): void	Visit a Role object and send the Visitor to visit all of its children
visitPermission	(permission: Permission): void	Visit a Permission object and see if its ID matches the ID provided to the visitor, if so then save the Permission as the result

Properties

Property Name	Type	Description
findById	String	ID of the Permission being looked for

Associations

Association Name	Type	Description
result	Permission	Permission that was found, or null

FindRoleByIdVisitor *implements AuthServiceVisitor*

This Visitor is used to iterate through the Entitlement composition tree until it finds a Role that has the same ID provided to the Visitor

Methods

Method Name	Signature	Description
getResult	():Role	Get the result of what the Visitor finds or null
visitRole	(role: Role): void	Visit Role and test whether or not the Role matches the provided ID, if it does not then send the Visitor to visit all the Roles children, this works

		because the Visitor is meant to be introduced at the top of the tree at the Root Role

Properties

Property Name	Type	Description
findById	String	ID of the Role being looked for

Associations

Association Name	Type	Description
result	Role	Role that was found, or null

InventoryVisitor *implements AuthServiceVisitor*

The Inventory Class is responsible for iterating through all of the objects in the Auth Service and printing out the data of each of the object

Methods

Method Name	Signature	Description
visitRole	(role: Role): void	Visit Role and print out the Role details then send the Visitor to visit the Roles children
visitPermission	(permission: Permission): void	Visit Permission and print out the Permission details
visitSmartCityAuthenticationService	(smartCityAuthenticationService: SmartCityAuthenticationService): void	Visit SmartCityAuthenticationService and use it for navigation
visitCredential	(credential: Credential): void	Visit Credential and print out the Credential details
visitUser	(user: User): void	Visit User and print out the User and each of their Credentials
visitResource	(resource: Resource): void	Visit Resource and print out the Resources details
visitAuthToken	(authToken: AuthToken): void	Visit authToken and print out the authToken details

---

Permission *extends Entitlement*

The Permission class represents what a User is able to do. If a User is associated with a Permission then they have access to the methods that also use the Permission, otherwise the User cannot use those methods. Permissions are the leaf nodes in the Entitlement composition. See Entitlement for additional class details inherited from super.

Methods

Method Name	Signature	Description
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

---

Resource *implements AuthServiceVisitorElement*

The Resource class represents and actual IoT device in the Smart City system

Methods

Method Name	Signature	Description
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

Properties

Property Name	Type	Description
id	String	An ID unique to from all other Resrouces
description	String	A friendly description for the Resource

ResourceRole *extends Role implements AuthServiceVisitorElement*

The Resource Role class connects a Resource and a Role via two associations it makes, one to a Role and one to a Resource. See Role for additional class details inherited from super.

Associations

Association Name	Type	Description
role	Role	The Role associated with Resource
resource	Resource	The Resource associated with the Role

Role *extends Entitlement implements AuthServiceVisitorElement*

Role extends the Entitlement subclass and represents a set of Entitlements that a User can be associated with while it can also contain Permissions as well as other Roles. Users who are associated to Role that contain Permissions have access to methods that require those Permissions. See Entitlement for additional class details inherited from super.

Methods

Method Name	Signature	Description
addEntitlement	(entitlement: Entitlement): void	Add an Entitlement (Role or Permission) to this Role
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

Associations

Association Name	Type	Description
children	HashMap<String. Entitlement>	All of the children Entitlements (Roles and Permissions) of this Role

---

SmartCityAuthenticationService *implements AuthServiceVisitorElement*

The Smart City Authentication Service is responsible for all of the functionality of authentication in the Smart City. It manages Roles, Permissions, Resources, Resource Roles, Users, Credentials and Auth Tokens as they relate to authentication in the Smart City system. It includes maps to store Users, Resources and Auth Tokens as well as a composite tree of Role, Permission and Resource Roles

Methods

Method Name	Signature	Description
getInstance	(): SmartCityAuthenticationService	singleton pattern getter
createResource	(id: String, description: String, authToken: String): void	Create a new Resource
createResourceRole	(id: String, roleId: String, resourceId: String, authToken: String): void	Create a new Resource Role, attach that Role to another Role and a Resource
createPermission	(id: String, name: String, description: String, authToken: String, authToken: String): void	Create a new Permission
createRole	(id: String, name: String, descripition: String, authToken: String): void	Create a new Role Entitlement
addPermissionToRole	(roleId: String, permissionId: String, authToken: String): void	Add an Entitlement (Permission/Role) to a Role
createUser	(id: String, name: String, authToken: String): void	Create a new User in the Auth Service
updateUserCredential	(userId: String, type: CredentialType, credential: String, authToken: String): void	Update a User Credential for the User ID provided
addRoleToUser	(userId: String, roleId: String, authToken: String):void	Add a Role Entitlement to the User whose ID is provided
login	(userId: String, credential: String):AuthToken	Log a User in by verify their credentials then issue an Auth Token. This method uses both user ID and credential string and is meant to be used with Username Password authentication
login	(credential: String): AuthToken	Log a User in by verify their credentials then issue an Auth Token. This method uses only the credential and is meant to be used with Biometric authentications
logout	(userId: String): void	Iterate through Auth Tokens to find every token related to the User ID provided and invalidate those tokens
checkAccess	(authToken: String, permissionId: String): void	Check to see if the Auth Token provided has access to the Permission ID also provided. Visitor Pattern is used to accomplish this.
printInventory	(authToken: String): void	Method to print the inventory of all the Entitlements, User, Auth Tokens and Credentials stored in the Authentication Service. This method utilizes the Visitor Pattern extensively.

acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	Standard visitor acceptor
findRoleById	(id: String, authToken: String): void	Find Role in Auth Service storage by Role ID. Visitor Pattern used.
findPermissionById	(id: String, authToken: String): void	Find Permission in Auth Service storage by Permission ID. Visitor Pattern used.

Properties

Property Name	Type	Description
rootRool	Role	Top level Entitlement, all Permissions and Roles live under this one, allowing for a Root or God-like admin user (which is used in the script)

Associations

Association Name	Type	Description
users	HashMap<String, User>	Map of all the Users in the Auth Service
resources	HashMap<String, Resource>	Map of all the Resources in the Auth Service
authTokens	HashMap<String, AuthToken>	Map of all the AuthTokens in the Auth Service

---

User

The User class represents a User in the Smart City Authentication Service. A Users ID matches the Users ID in the Smart City Model Service and once Users are created in the Model Service, they're also created in the Authentication Service. Users include a map of Credentials that allows them to authenticate under multiple various authentication methods as well as a map of entitlements that allow evaluations of the Users individual Roles and Permissions.

Methods

Method Name	Signature	Description
verifyCredentialString	(credentialString: String): Boolean	Iterate through each credential associated with this User and see if any of them match the non hashed credential string passed in
findPermissionById	(id: String): Permission	Use the visitor pattern to try to find if the User has access to the Permission ID passed in
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	standard visitor pattern acceptor

Properties

Property Name	Type	Description
id	String	An ID unique to all the stored Users
name	String	A friendly name for the User

Associations

Association Name	Type	Description

credentials	HashMap<String, Credential>	All of the possible Credentials the user could use to Auth against
entitlements	HashMap<String, Entitlement>	All of the Entitlements of the User (Roles and Permissions)

CredentialTypeEnum

Provides a selection of different types of a credentials a User can have associated with them.

Enum Name
BIOIMETRIC
PASSWORD

AuthTokenStateEnum

Provides a selection of different states an Auth Token can be in help to determine if its valid or not.

Enum Name
ACTIVE
INACTIVE

AuthServiceVisitorElement Interface

Provides an Interface for Visitor Elements to adhere to that contains methods.

Methods

Method Name	Signature	Description
acceptVisitor	(authServiceVisitor: AuthServiceVisitor): void	text

---

AuthServiceVisitor Interface

Provides an Interface for Visitors to adhere to that contains methods.

Methods

Method Name	Signature
visitUser	(user: User): void
visitResource	(resource: Resource): void
visitAuthToken	(authToken: AuthToken): void
visitRole	(role: Role): void
visitPermission	(permission: Permission): void

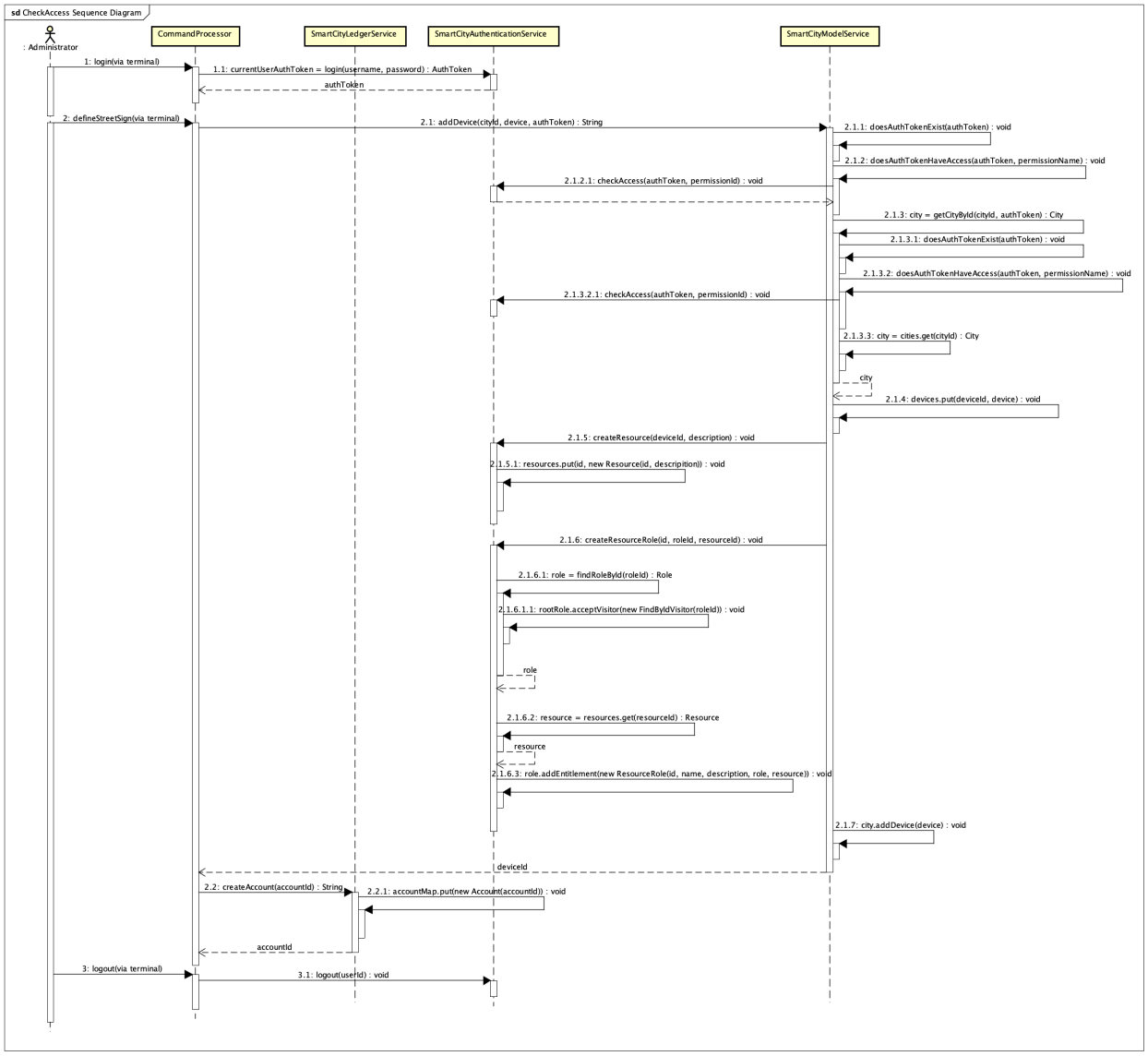
visitSmartCityAuthenticationService	(smartCityAuthenticationService: SmartCityAuthenticationService): void
visitCredential	(credential: Credential): void

Implementation Details

General Implementation Details

The majority of the functionality in the Smart City Authentication Service is dedicated to the management of the Authentication Service's authentication and user data. Features like adding User, Roles and Permissions give the administrator the ability to create a flexible authentication system. Few features in the Authentication Service exist without a concern of managing the Authentication Service itself. One of these features is the check access feature. Security in the Smart City System is implemented by way of the check access feature in the Authentication Service. Nearly every method in the Smart City Model Service calls the checkAccess() method in the Authentication Service in order to validate whether or not the caller of that Model Service method is provisioned to execute the method. This pattern is at the core of how the Authentication Service interfaces with the rest of the Smart City System and it's what keeps the Smart City System secure.

The below sequence diagram shows the interactions involved when an Administrator adds a new Device to the Smart City System and the Auth Service is required the check the Administrators access privileges. This diagram is also available in a larger version in the submission folder.

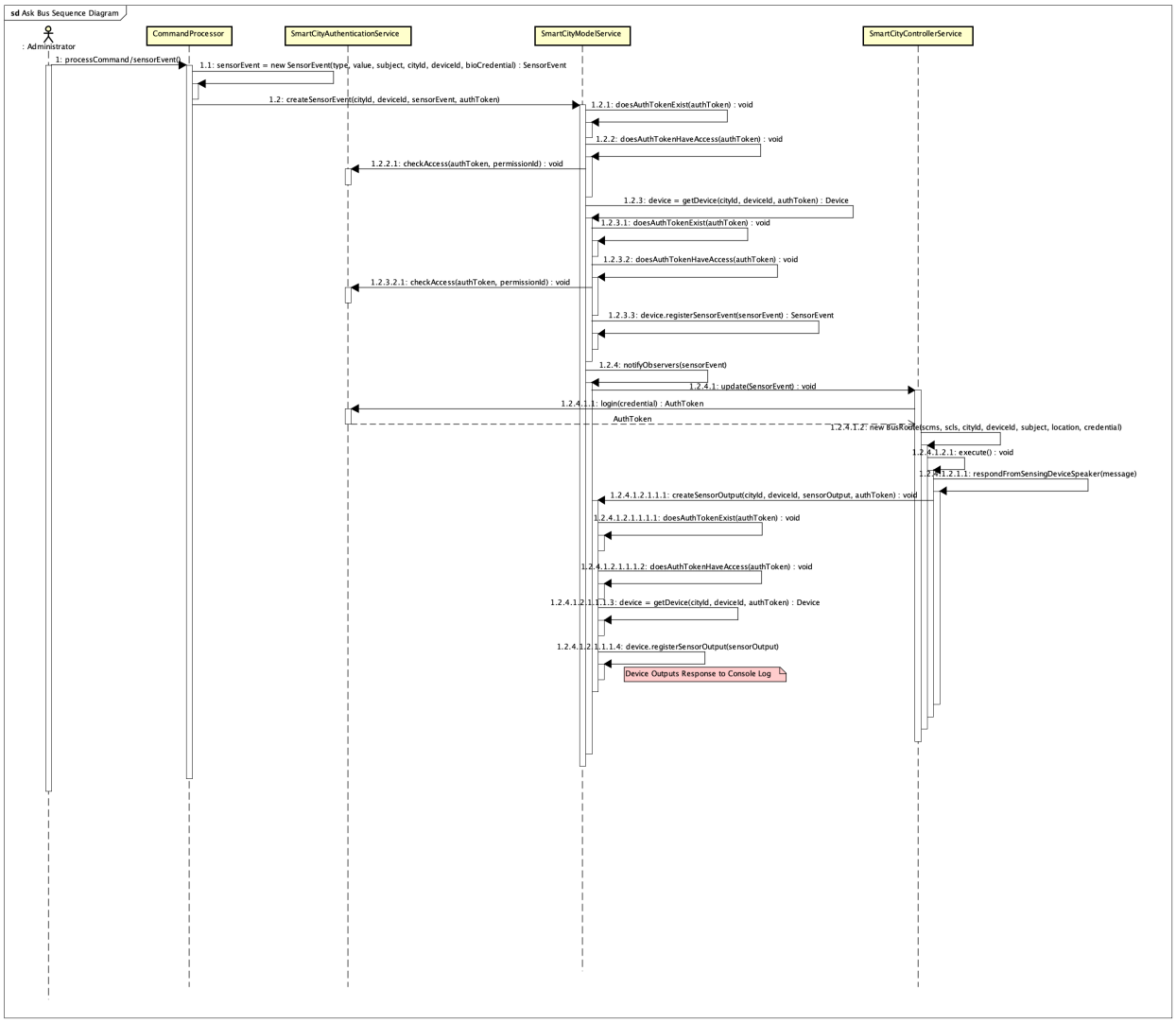


Implementation of the Visitor Pattern in the Authentication Service

The visitor pattern plays a key role in enabling the Smart City Authentication Service to efficiently find permissions its asked to look for when checking to see if a Registered User is authorized to access a certain method in the Smart City Model Service. Through the checkAccess() method in the Authentication Service, clients can provide an Auth Token and a Permission ID in order to see if the user associated with that Auth Token has access to the Permission whose ID was provided. The visitor pattern is used to traverse through the AuthToken's composite structure of Roles and Permissions, verifying that each Permission of the AuthToken either does or does not provide the User with the required access.

The below sequence diagram shows the interactions involved when a Person interacts with a Bus device in the Smart City System to create a Sensor Event and the Controller is then forced

to access the Model Service on the Persons behalf by using the Auth Service. This diagram is also available in a larger version in the submission folder.



Implementation of the Singleton and Composite Patterns

The singleton pattern is used extensively in Assignment 4. Specifically, the Model, Controller and Authentication services all use the singleton pattern. The Authentication Service implements the singleton pattern to ensure there is one instance of the Authentication Service that is sure to act as the single source of truth for all of the Users and their privileges in the Smart City System. The composite pattern is also used in the Authentication Service. The Authentication Services stores Permission data in a tree structure that is home to both Permissions and Roles that then may contain more Roles or Permissions. The composite pattern is implemented in order to allow both Roles and Permissions to reside the same data structure. The Visitor Pattern is also used extensively in the print inventory feature where each object in the Smart City Authentication Service is visited and made to print out information about itself.

The print inventory feature's navigation is done using a combination of navigation both the in the Smart City Authentication Service and in the InventoryVisitor.

Exception Handling

There are two main types of exceptions related to the Smart City Authentication Service. The first exception is the InvalidAccessTokenException. This exception is thrown when a method from the Smart City Model Service calls the checkAccess method in the Authentication Service and the Authentication Service determines either that the Auth/Access Token doesn't exist for that User or is in some other way invalid. The second and more generic exception is the AuthServiceException and this exception is thrown when an error occurs in the Authentication Service but does not qualify to be thrown under the InvalidAccessTokenException.

AuthServiceException *extends Throwable*

This exception is thrown as a generic exception for faults in the Smart City Authentication Service.

Methods

Method Name	Signature	Description
getAction	text	text
getReason	text	text

Properties

Property Name	Type	Description
action	String	text
reason	String	

InvalidAccessTokenException *extends Throwable*

This exception is thrown as a result of faults in the Smart City Authentication Service related to an Auth Token being used that is either invalid, expired, inactive or doesn't have access to the required Permission.

Methods

Method Name	Signature	Description
getAction	text	text
getReason	text	text

Properties

Property Name	Type	Description
action	String	text
reason	String	

Testing

Testing was done using a modified version of the provided test script. The original script was not included for a few reasons, it required quite a few changes to fix inconsistencies and in section we were directed to use that script as an example starting point rather than a test utility like we had done in previous assignments. The test script used to build and test my assignment 4 is an extension of the sample provided and is named Assignment_4_custom_sample_script.script in the submission folder.

Risks

I think the main Risk of this system is that, despite the composite tree structure being a clever way to organize Entitlements, it seems like it would be hard to manage without having a graphical user interface. Another risk is that the methods requiring Permission have the Permission ID hard coded in the method itself. In order to effectively use the Auth Service, I think these would need to be dynamic somehow. It might make more sense to have a configuration file that loads these Entitlement parameters then rather than a CLI interface.

System Changes

I didn't have very many changes for the Assignment 4. I did find the concept of ResourceRoles pretty confusing however. Because of this, my implementation of ResourceRoles might not be as robust as expected from graders and Eric. I'm still not entirely sure how ResourceRoles are supposed to be considered in whether or not someone has access to a method, but I implemented them as best I could based off of the requirements documents.