CSCI E-97 Software Design 11/14/2020

Smart City Authentication Service, Design Document

Date: 11/14/2020

Author: Loi Cheng

Reviewer(s): Benjamin Basseri

Introduction

The city of the future can be automated by the Smart City Software System. through AI-powered Internet of Things (IoT) devices, smart City allows city administrators to fully automate a city. Devices include cameras, microphones, robots, and other devices. Sensors monitor the condition of people and devices. Parking spots automatically charge fees to the user's blockchain account. Robots help residents clean the city, respond to emergencies, and fight terrorists.

Caption: People and Devices in the Smart City
(Source https://internetofbusiness.com/global-smart-city-platform-market/)

Overview

The implementation of a Smart City Authenticator is described. The authenticator manages permissions for users for each method available across the smart city api. It allows users to automatically login with their face or voice to interact with IoT devices.

Requirements

This section defines the requirements for the Smart City Authentication Service.
The service should support the following functions:

The Authentication Service API supports the following functions:

1. Creating Resources:
    a. A Resource represents a physical and logical entity, for example, an IoT Device.
    b. A Resource has a unique identifier and a description.
2. Creating Permissions:
    a. Permissions represent an authorization required to access a resource or function of the Smart City system.
    b. Permissions have a unique id, name, and description.
    c. A User may be associated with zero or more permissions.
3. Creating Roles:
    a. Roles are composites of Permissions.
    b. Roles provide a way to group Permissions and other Roles.
    c. Like Permissions, Roles have a unique id, name, and description.
    d. Users may be associated with Roles, where the user has all permissions included in the Role or sub Roles.
    e. Roles help simplify the administration of Users by providing reusable and logical groupings of Permissions and Roles.
4. Creating Users:
    a. Users represent persons of the Smart City system.
    b. Users have an id, a name, and a set of Credentials. Credentials may include a username/password, voiceprint, faceprint, and other biometric identities. Hash the credentials to secure their use.
    c. Users are associated with 0 or more entitlements (i.e., Roles or Permissions).
5. Authentication
    a. The Authentication process provides users AuthTokens that can then be used to access restricted Service Methods.
    b. If authentication fails, an Authentication Exception should be thrown.
    c. If authentication succeeds, an AuthToken is created and returned to the caller.

d. The AuthToken binds the User to a set of permissions that can be used to grant or deny access to restricted methods.
e. AuthTokens can timeout with inactivity.
f. AuthTokens have a unique id, an expiration time, and a state (active or expired).
g. Auth tokens are associated with a User and a set of Permissions.
    i. Login:
        1. Login accepts a User's credentials (username, password).
        2. Validate that the username exists, and then that the hash of the password matches the known hashed password.
    ii. VoicePrint
        1. Voiceprint supports authentication through voice recognition
        2. Simulate a voiceprint using a string in this format: " voice-print= voiceprint- <username>". For example, the voiceprint for Jane is " voice-print='voiceprint-jane' ".
        3. The voice print signature is sufficient for identifying and authenticating a user.
    iii. FacePrint
        1. Faceprint supports authentication through face recognition
        2. Simulate a faceprint using a string in this format: " face-print='faceprint-<username>". For example, the faceprint for Jane is "face-print='faceprint-jane'" .

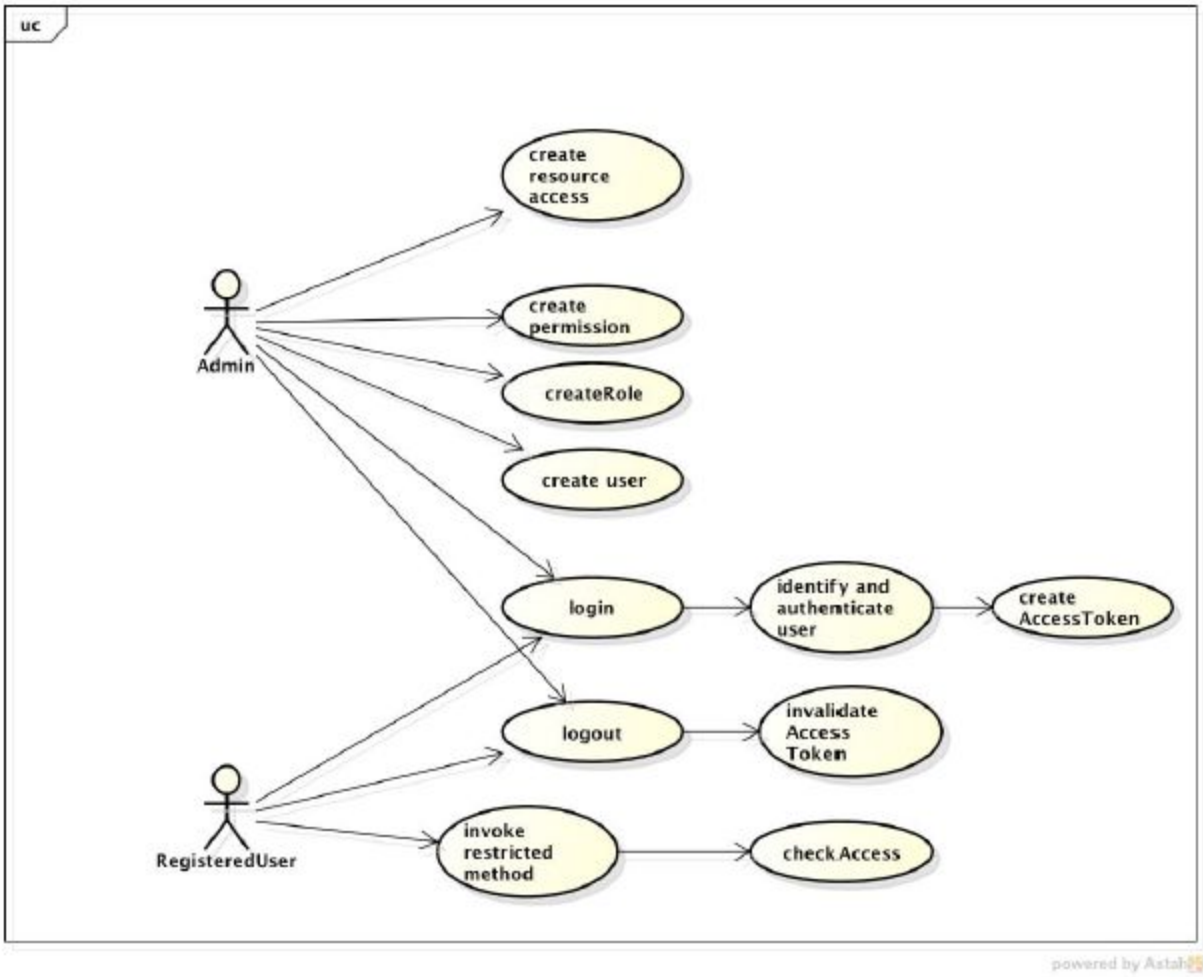
6. Logout:
    a. Logout marks the given Auth Token as invalid.
    b. Subsequent attempts to use the AuthToken should result in an InvalidAuthTokenException.
7. Smart City Model Service:
    a. All methods defined within the Model Service should accept an AuthToken
    b. Each method should validate that the AuthToken is non-null and non-empty.
    c. The method should pass the AuthToken to the Authentication Service with the required permission.
    d. The Authentication Service should check to make sure that the AuthToken is active and within the expiration period. Then, check that the user associated with the AuthToken has the permission required by the method.
    e. The Authentication Service should throw an AccessDeniedException or InvalidAccessTokenException if any of the checks fail.

Design Patterns:

1. Use the Visitor Pattern to:
    a. support traversing the Authentication Service objects to provide an inventory of all Users, Resources, Accesses, Roles, and Permissions.
    b. Checking for access
2. Use the Singleton Pattern to return a pointer to an implementation of the Authentication Service.
3. Use the Composite Pattern to manage the whole part relation of Roles, Resource Roles, and Permissions.

Use Cases:

The Authentication Service will assist in controlling access to the Model Service interface. The Authentication Service provides a central point for managing Users, Resources, Permissions, Roles, ResourceRoles, and Auth Tokens.



Caption: Use Case Diagram for Authentication Service

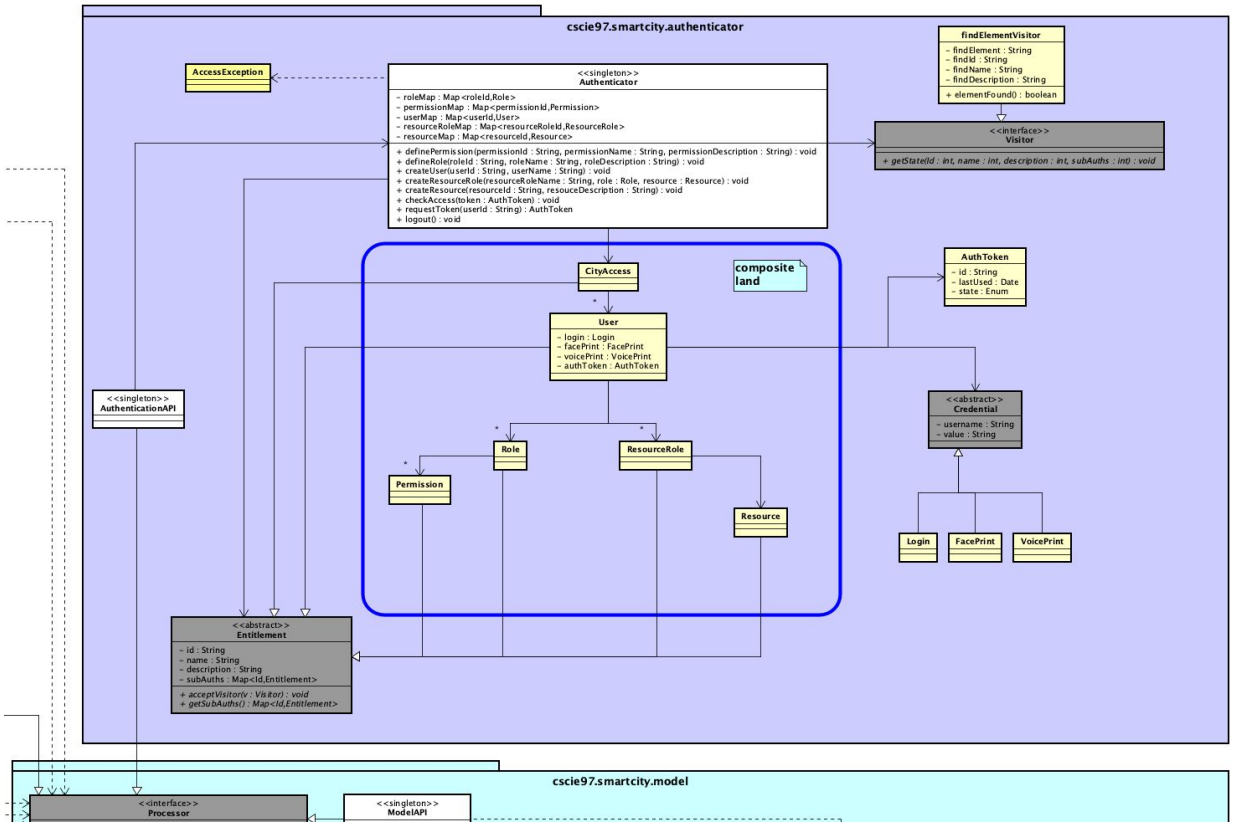
The Authentication Service supports two primary actors: Administrator and Registered User. The Administrator is responsible for managing the Resources, Permissions, Roles, and Users maintained by the Authentication Service. The Administrator also provisions Cities and other entities within the Smart City system and has full access to all Smart City System resources. Voice or face prints identify city inhabitants.

Inhabitants use voice commands to interact with the Smart City System. The Authentication Service supports identifying and authenticating users using the user's voice or face print. When a Consumer issues a voice command, the Controller service sends the voiceprint and faceprint to the Authentication Service. The Authentication Service finds the user with a matching biometric identifier. It returns an Authentication Token, then passes the authentication token to the Model Service methods on behalf of the Inhabitants. In this way, only persons with the appropriate permissions can control IoT devices within the cities.

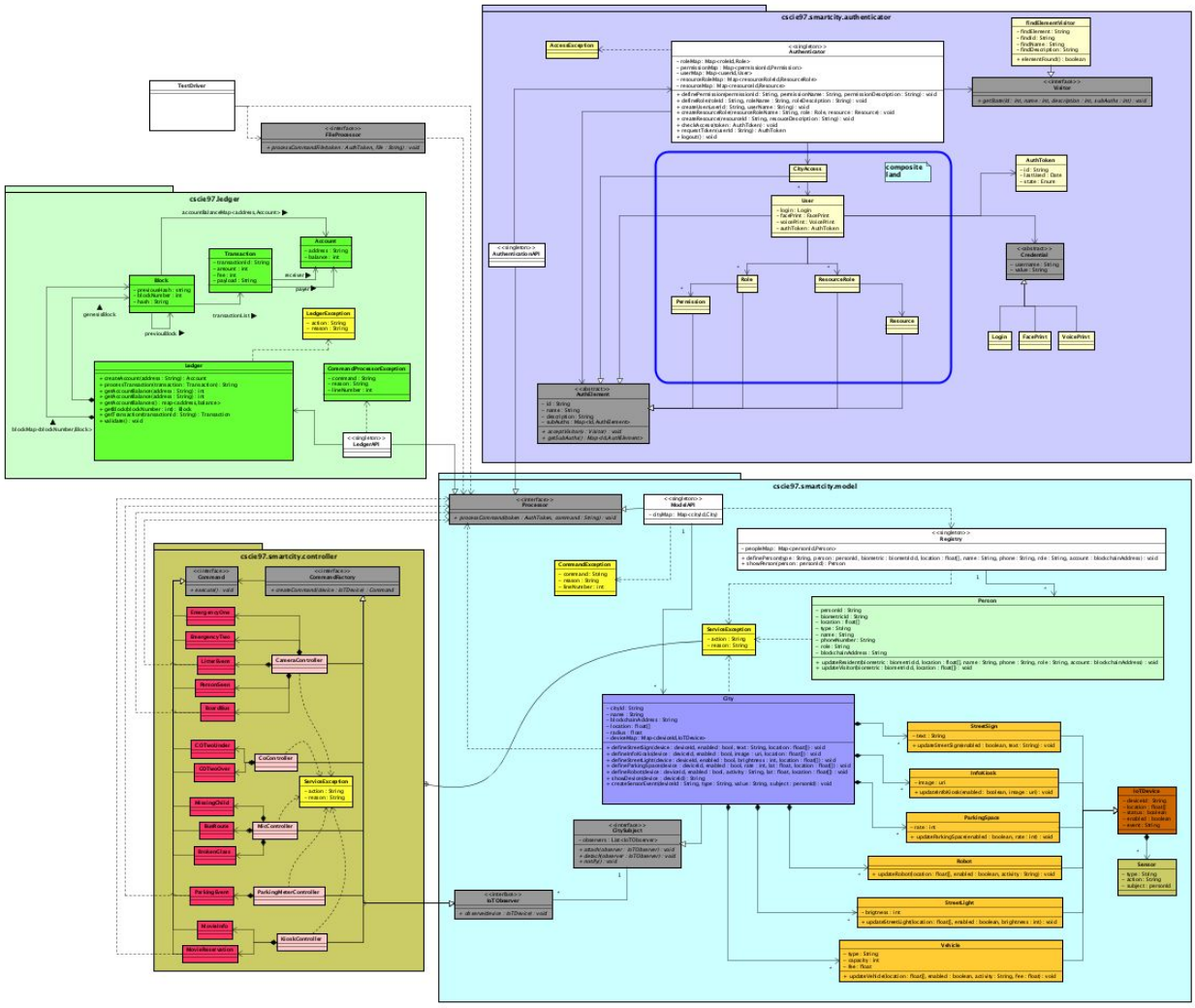
Implementation

Class Diagram

The following class diagram defines the implementation classes contained within the package “cscie97.smartcity.authenticator”.



Caption: Smart City Authenticator Class Diagram



Caption: Entire Smart City System Class Diagram

Class Dictionary

AuthenticationAPI Class (singleton)

This class provides an interface for a user to input commands to the authentication service. Only one API is created.

Method Name	Signature	Description
proccessCommand	(command:String, lineNumber:int) : void	Processes a user input line

Authenticator Class (singleton)

This class separates the AuthenticationAPI and the Model Services from all the other authentication related classes. Outside methods use this class to manipulate objects within the authentication service. This is a singleton class, there is only one created.

Property Name	Type	Description
bootStrapMode	boolean	Switch to enable or disable bootstrap mode, where no permission is required for services. On startup it is true and set to false after bootstrap finishes.

Association Name	Type	Description
adminCity	AccessCity	An admin in this has access to all other cities. This is only set once on start up. It does not have an equivalent city in the model service.
token	AuthToken	Store the token of the currently logged in user.
accessCity	AccessCity	Store the pointer of the current city under use
entMap	Map<String, Entitlement>	Stores pointers to all the entitlements created.

accessCityMap	Map<String, Entitlement>	Stores pointers to all the AccessCity created

Method Name	**Signature**	**Description**
login	(commands List<String>) : void	Logs in the user by getting a valid token to use.
requestToken	(userId, String password) : AuthToken	Private helper to the public login method. It gets the token based on the provided credentials
authenticate	(method:String, identifier: String):void	Checks the access of a user to the method. Throws error if no access.
checkAccess	(elem: EntType, id: String, name: String, resourceId String)	Checks if the token user has the entitlement in the composite tree. The visitor pattern is used.
logout	(): void	Log out the user by marking token as expired
setCityAccess	(String cityId):void	Changes the current city in use
definePermission	(id: String, name: String, description: String):void	Create a Permission object
defineRole	(id: String, name: String, description: String):void	Create a Role object
createAccessToCity	(id: String, name: String, description: String):void	Create a AccessCity object
createUser	(id: String, username:String):void	Create a User object
createResource	(id: String, name: String, description: String):void	Create a Resource object
createResourceRole	(id: String, name: String,	Create a ResourceRole object

	description: String):void	
addSub	(parent: String, child: String):void	Add an Entitlement object into the SubAuths maps of another Entitlement object. This builds the composite tree
addUserCredential	(userId:String, credentialType:String, value:String):void	Add a credential to a user - eg. password, faceprint, voiceprint
showEntitlements	(userId:String):void	Show all the entitlements attached to user
showAllEntitlements	():void	Show all the entitlements for all users

AuthToken Class

This object enables a user to pass authentication to user model services.

Property Name	**Type**	**Description**
useLimit	Integer	A specified amount of commands the token is good for, then the token will expire afterwards
userId	String	The user id to the user of this token
lastUsed	Timestamp	The last time the token was used
state	Enum	The state of the token- active or expired
uuid	UUID	The unique identifier for the token. This is auto-generated
counter	Integer	The counter keeps track of how many times the token has been used.

Method Name	Signature	Description
updateTime	():void	Updates the last time the token was used
expire	():void	Mark the token as expired
tokenUsed	():void	Increments the counter by one use.

Credential Class

This class stores the password, faceprint, and voiceprint of a user, in hashed format

Property Name	Type	Description
userId	String	The user id
passHash	byte[]	The hashed password
faceHash	byte[]	The hashed faceprint
voiceHash	byte[]	The hashed voiceprint

Association Name	Type	Description
userToken	AuthToken	The token

Method Name	Signature	Description
setPass	(oldPass:String, newPass:String):void	Sets the user password, or changes it if it already exists

setFace	(oldPass:String, newPass:String):void	Sets the user faceprint, or changes it if it already exists
setVoice	(oldPass:String, newPass:String):void	Sets the user voiceprint, or changes it if it already exists
updateToken	(user:String, pass:String)	If pass is ok, create a new token if existing is expired, attach it to the user's Credential

Entitlement Class

The entitlement class is the building block for the other classes like Resource. Multiple entitlement objects are linked together to form a composite pattern, which describes the permissions for a given user.

Property Name	**Type**	**Description**
cityId	String	The city the entitlement is associated with
id	String	The id of the entitlement
name	String	The name of the entitlement
description	String	The description of the element
elem	Enum	The type of the entitlement - Resource, Role, ResourceRole, Permission, User, AccessCity

Association Name	**Type**	**Description**
subAuths	Map<String, Entitlement>	A map of the entitlements directly under this class

Method Name	**Signature**	**Description**

acceptVisitor	(visitor: Visitor): void	Enables the visitor to retrieve info on this class. This applies the visitor pattern requirement.
addSubAuth	(id: String, ent: Entitlement)	Remove an observer from the list of observers

Resource Class

The Resource is a type of entitlement in the composite It represents a single entity, like an IoTdevice in the city. It extends all the properties and methods in the Entitlement Class. It may take on additional methods and properties in the future. The differences between Resource and Entitlement are specified below.

Property Name	Type	Description
elem	Enum	The type of the entitlement - Resource

Role Class

The Resource is a type of entitlement in the composite. It represents a single entity, like an IoTdevice in the city. It extends all the properties and methods in the Entitlement Class. It may take on additional methods and properties in the future.

Property Name	Type	Description
elem	Enum	The type of the entitlement - Role

ResourceRole Class

The Resource is a type of entitlement in the composite. It represents a single entity, like an IoTdevice in the city. It extends all the properties and methods in the Entitlement Class. It may take on additional methods and properties in the future.

Property Name	Type	Description
elem	Enum	The type of the entitlement - ResourceRole

Permission Class

The Permission is a type of entitlement in the composite. It represents a single entity, like an IoTdevice in the city. It extends all the properties and methods in the Entitlement Class. It may take on additional methods and properties in the future.

Property Name	Type	Description
elem	Enum	The type of the entitlement - Permission

User Class

The User is a type of entitlement in the composite. It represents a single entity, like an IoTdevice in the city. It extends all the properties and methods in the Entitlement Class. In addition to the entitlement methods and properties, the user also has credential properties

Property Name	Type	Description
elem	Enum	The type of the entitlement - User

Association Name	Type	Description
creds	Credential	The credentials - password, voiceprint, faceprint, of the user

AccessCity Class

AccessCity is a type of entitlement in the composite. It is the identifier for the city in the authentication service. It is at the root of the composite tree and holds the users of the city as its leaves.

Property Name	Type	Description
elem	Enum	The type of the entitlement - AccessCity

Visitor Interface

This interface is provided for the class sending the visitor and the class accepting the visitor to interact with the visitor.

Method Name	Signature	Description
getState	(cityId: String, elem:EntType, id:String, name:String, desc:String, subAuths:Map<>):void	The entitlement calls this method to give the visitor all the properties that it has
outcome	():void	The outcome of the search, true if the entitlement was found
visitComplete	():Boolean	True if the visitor has completed the visit.

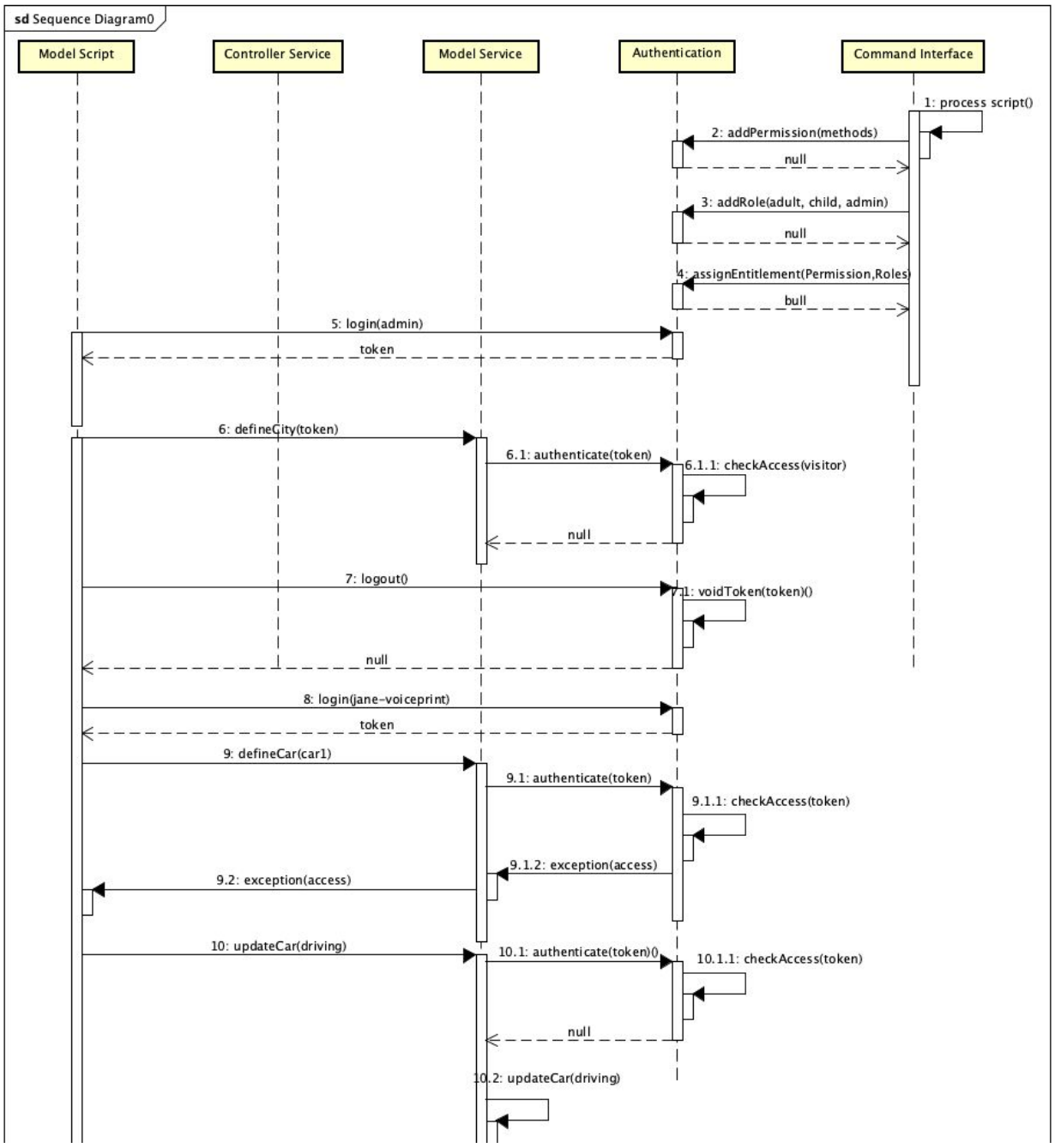
FindEntitlementVisitor Class

This class visits the entitlements associated with a user to retrieve some specific properties about them. This class meets the Visitor design pattern requirement. In addition to the methods in the Visitor interface, it also has these properties below

Property Name	Type	Description
findElem	Enum	The entitlement type to find
findName	String	The name of the entitlement to find
findId	String	The id of the entitlement to find
findResourceName	String	The resource name to find
complete	Boolean	Marks if the visit to the composites is complete
fullVisit	Boolean	If true, continue to visit other entitlements ev

Implementation Details

Below is a sequence diagram demonstrating an implementation:



Caption: Sequence Diagram, general flow

In the sequence diagram, when the services first start, a set of initial scripts creates and assigns permissions to a super admin, who has control of everything. Once the initial script is complete, the

system permissions kick in and auth tokens are required to access the methods. The super admin logs in and gets a token. The admin then creates a city, and then the model service authenticates the admin's token to check if creating a city is allowed. The authenticator finds the access, and allows the routine to proceed, and then the city is created. The admin then logs out, and the token is no longer valid.

Jane, a general user, automatically logs in with her voice print as she tries to define a new IoT car. The model service checks her token, and it does not have permission to define-car, and throws an access exception to stop the routine. Jane then does an update-car to drive it, and the model service checks with the authenticator again to see if her token has access to drive the car. It does not throw any exceptions so it passed, and the model service continues and sets up the car for driving.

Exception Handling

AuthException Class

This is a general class used to handle most exceptions that occur in the authentication service

Property Name	Type	Description
action	String	The action that caused the exception
reason	String	The reason that caused the exception

Method Name	Signature	Description
AuthException	(action:String, reason:String):void	Creates an AuthException

Testing

Implement a similar TestDriver class based on the city model service, that implements a main() method. The main() method can call these methods:

FileProcessor.processCommandFile(file:String)
ModelApi.processCommand(command:String, -1)

The main method can receive a command file, or allow a user to enter commands in a console if no file is specified. The TestDriver class should be defined within the package “cscie97.smartcity.test”.

- The program and tests should compile with this command executed in the base directory:

#run as shell script in linux terminal
javac cscie97/ledger/*.java cscie97/smartcity/model/*.java cscie97/smartcity/controller/*.java cscie97/smartcity/shared/*.java cscie97/smartcity/test/*.java cscie97/smartcity/authenticator/*.java

- The test should run with the command:

java -cp . cscie97.smartcity.test.TestDriver all.script.txt > all.out.txt

all.script.txt should list the name of all the script files to process
all.out.txt should have the output of the script

NOTE: the absolute filepath of the java files cannot contain any spaces! Example

/Users/smartcity/assignment4/ ...ok
/Users/smartcity/assignment 4/ ...no!

Risks

The authentication service is at high risk of being hacked. Anyone that knows the login name and password of any admins can create significant damage to the system. Logins should implement multi-factor authentication. Admin access should be restricted to only several physical locations and ip addresses, in order to prevent remote hacking, so that hackers would need to physically infiltrate the control center to gain access.