# Authentication Service Design Document

Date: 10/26/2020
Author: Christopher Jones
Reviewer(s):

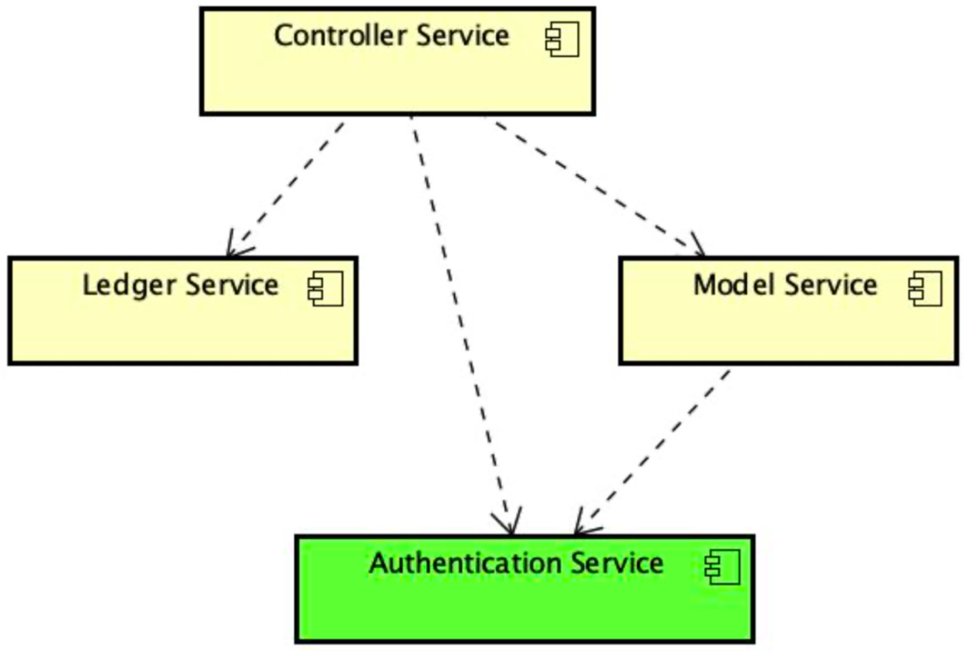
- Allyson Bieryla
- Jaya Johnson

## Introduction

This document defines the design for the Smart City Authentication Service.

## Overview

The Smart City Authentication Service is responsible for authenticating users, resources, and services with respect to the Smart City. It allows a specified group of users to gain predefined access through roles and permissions to respond to certain stimuli within the Smart City. Additionally, it prevents users from accessing certain city features and functionalities that they shouldn't need access to. The authentication functionality is also accessible across multiple packages such that any given city can be managed by exactly one instance of the Authentication Service.



## Requirements

The section proves a summary of the requirements for the Authentication Service.

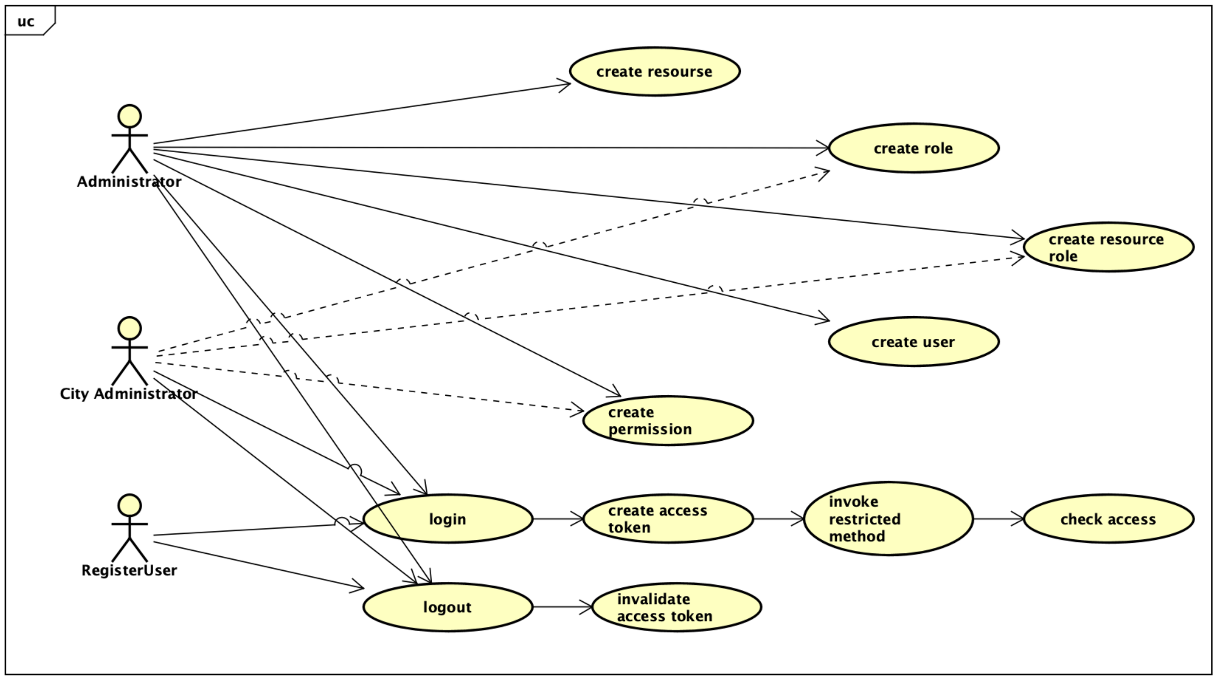
The Smart City Authentication Service is primarily responsible for controlling access within a given city (the Model Service). It provides a central point for managing Users, Resources, Permissions, Roles, ResourceRoles, and Auth Tokens. Similar to the Controller Service, the Authentication Service monitors the actions of the Model Service and its inhabitants, and in turn, creates its own internal entities to reflect them. So, should a stranger or individual defined outside of the Smart City try to access elements within the Model Service, they will not be able to since they won't exist within the Authentication Service.

The Authentication Service is made up of the following elements:

- Access Control Constrainers
    - Roles
        - ResourceRoles
    - Permissions
    - Credentials
        - Usernames/Passwords
        - Biometric Values (Voiceprint, Faceprint, etc.)
- Access Control Providers
    - Auth Tokens
- Pseudo-Physical Entities
    - Users
    - Resources
- Dependent Services
    - Model Service
    - Controller Service

## Use Cases

The following Use Case descripts the use cases supported by the Authentication Service.



There are 3 types of actors to consider:

- Primary Administrator
- City Administrator
- Registered User

**Primary Administrators** are often the first user established by the Authentication Service. They are provided administration roles that not only allow them to provide access to the city, but to build up entities within the Authentication Service itself. It can create Users, Resources, and even assign Roles and Permissions to other Users. Primary administrators should have access to all features and functionalities within a city and the ability to override any other users Roles and Permissions.

**City Administrators** are often the users defined after the primary administrators and the establishment of a city. These users are similar to the primary administrators; however, all of their access control applies to a given resource or set of resources applied by the primary administrator. While this is not always the case, the city administrator assists its city's inhabitants access particular devices they may need, and they allow the devices within the city to freely interact with individuals within the right role constraints.

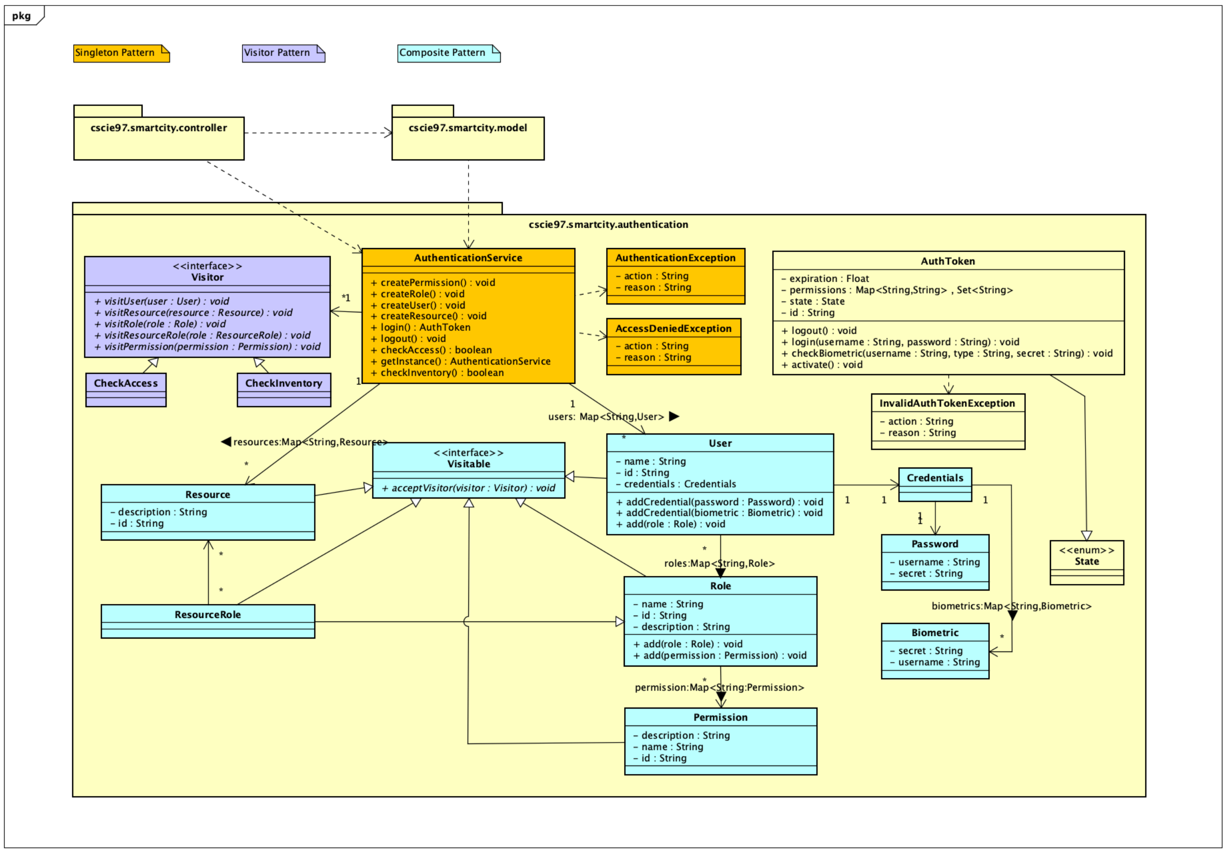
**Registered Users** are usually the inhabitants of the city and have limited permissions within the city. These users are not able to provide access to create or update anything, and are often accompanied by the city administrator to do most tasks. Among the tasks provided to these users, they are all simply interacting with the devices/resources within the city (i.e. asking for directions, movie recommendations, etc.). Registered Users can be split further in roles such as adult and children, whereas a child should have the least amount of access amongst all users.

## Implementation

This section of the document will describe the implementation details for the Authentication Service.

## Class Diagram

The following class diagram defines the classes defined in this design.



pkg
Singleton Pattern
Visitor Pattern
Composite Pattern
cscie97.smartcity.controller
cscie97.smartcity.model
cscie97.smartcity.authentication
<<interface>>
Visitor
+ visitUser(user : User) : void
+ visitResource(resource : Resource) : void
+ visitRole(role : Role) : void
+ visitResourceRole(role : ResourceRole) : void
+ visitPermission(permission : Permission) : void
CheckAccess
CheckInventory
AuthenticationService
+ createPermission() : void
+ createRole() : void
+ createUser() : void
+ createResource() : void
+ login() : AuthToken
+ logout() : void
+ checkAccess() : boolean
+ getInstance() : AuthenticationService
+ checkInventory() : boolean
*1
1
AuthenticationException
- action : String
- reason : String
AccessDeniedException
- action : String
- reason : String
AuthToken
- expiration : Float
- permissions : Map<String,String> , Set<String>
- state : State
- id : String
+ logout() : void
+ login(username : String, password : String) : void
+ checkBiometric(username : String, type : String, secret : String) : void
+ activate() : void
InvalidAuthTokenException
- action : String
- reason : String
1
users: Map<String,User> ▶
*
◀ resources:Map<String,Resource>
*
<<interface>>
Visitable
+ acceptVisitor(visitor : Visitor) : void
Resource
- description : String
- id : String
*
*
ResourceRole
User
- name : String
- id : String
- credentials : Credentials
+ addCredential(password : Password) : void
+ addCredential(biometric : Biometric) : void
+ add(role : Role) : void
1
1
Credentials
1
1
Password
- username : String
- secret : String
<<enum>>
State
*
roles:Map<String,Role>
Role
- name : String
- id : String
- description : String
+ add(role : Role) : void
+ add(permission : Permission) : void
biometrics:Map<String,Biometric>
*
Biometric
- secret : String
- username : String
*
permission:Map<String:Permission>
Permission
- description : String
- name : String
- id : String

## Class Dictionary

This section specifies the class dictionary for the class.

### AccessDeniedException Class

The AccessDeniedException is thrown whenever AuthToken creation fails due to incorrectly provided credentials (i.e. a bad username/password, mismatch biometrics, etc.).

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| getAction | getAction():String | This returns the action performed that caused the exception |
| getReason | getReason():String | This returns the reason why this action was a problem |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| action | action:String | The action that was performed |
| reason | Reason:String | The reason for the exception |

### Authentication Class

The Authentication class is the primary class used to control access to the Model Service. It creates internal copied entities and resources to represent the inhabitants within the city it monitors that it uses to simulate interactions and ultimately determine who has access and who does not.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| getInstance | getInstance():Authentication | This allows all dependent services to access the same instance of the Authentication service |
| login | login(username:String, password:String):AuthToken | This checks whether a user’s username, password, and biometrics match what |

| | | |
|---|---|---|
| | | it has on file. If it does, it grants them an AuthToken. |
| checkInventory | checkInventory(user:User):boolean | This uses the CheckInventory class to check whether it has record of the given entity |
| checkAccess | checkAccess(username:String):boolean | This uses the checkAccess class to check whether the identifiers of the given entities match what it has on file |
| createPermission | createPermission(id:String, name:String, description:String):void | Creates a new permission and saves it. If it already exists, throw an AuthenticationException. |
| createRole | createRole(id:String, name:String, description:String):void | Creates a new role and saves it. If it already exists, throw an AuthenticationException. |
| createUser | createUser(id:String, name:String):void | Creates a new user and saves it. If it already exists, throw an AuthenticationException. |
| createResource | createResource(id:String, description:String):void | Creates a new resource and saves it. If it already exists, throw an AuthenticationException. |
| createResourceRole | createResourceRole(id:String, role:String, resource):void | Creates a new resource role and saves it. If it already exists, throw an AuthenticationException. |
| addCredential | addCredential(userID:String, type:String, secret:String):void | Adds a credential to a user. If this cannot be done, throw an AuthenticationException. |

| | | |
|---|---|---|
| addRole | addRole(userID:String, role:String):void | Adds a role to a user. If this cannot be done, throw an AuthenticationExceptio n. |
| addResourceRole | addResourceRole(userID:String, roleID:String):void | Adds a resource role to a user. If this cannot be done, throw an AuthenticationExceptio n. |
| addPermission | addPermission(roleID:String, permissionID:String):void | Adds a permission to a user. If this cannot be done, throw an AuthenticationExceptio n. |

**Properties**

| **Property Name** | **Signature** | **Description** |
|---|---|---|
| authInstance | authInstance:Authentication | This allows all dependent services to access the same instance of the Authentication service |

**Associations**

| **Association Name** | **Signature** | **Description** |
|---|---|---|
| permissions | permissions:HashMap<String,Permission> | A mapping of established permissions |
| resources | Resources:HashMap<String,Resource> | A mapping of established resourses |
| roles | Roles:HashMap<String,Role> | A mapping of established roles |
| resourceRoles | resourceRoles: HashMap<String,ResourceRole> | A mapping of established resourse roles |
| users | users: HashMap<String,User> | A mapping of established users |
| tokens | tokens: HashMap<String,AuthToken> | A mapping of established auth tokens |

## AuthenticationException Class

The AuthenticationException is thrown whenever a reference error occurs within the Authentication Service. That is, if attempting to apply an action to a user, resource, role, permission, etc. that does not exist, this exception will be thrown in response.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| getAction | getAction():String | This returns the action performed that caused the exception |
| getReason | getReason():String | This returns the reason why this action was a problem |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| action | action:String | The action that was performed |
| reason | Reason:String | The reason for the exception |

## AuthToken Class

AuthTokens are provided upon successful logins into the Authentication Service. Tokens store a collection of permissions that the owner can then use to invoke methods that fit within their scope. Tokens have an expiration time that can mark it invalid, or alternatively, a logout will do the same.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| activate | activate():void | Initial sign-in provided that the given credentials are correct |
| login | login(username:String, password:String) | This checks whether a user's username, and password match what it has on file. If |

| | | |
|---|---|---|
| | | it does, it grants them an AuthToken. |
| checkBiometric | checkBiometric(username:String, type:String, secret:String) | This checks whether a user's biometrics match what it has on file. If it does, it grants them an AuthToken. |
| logout | logout():void | Invalidates the token |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| id | Id:String | The unique ID of the token |
| expiration | expiration:long | The time that the token will expiration |
| permissions | permissions:HashSet<String> | The permissions that the token provides |

**Associations**

| Association Name | Signature | Description |
|---|---|---|
| state | State:State | Whether or not the token is active or invalid |

## Biometric Class

A biometric object represents a physical credential related to an individual user. These objects can represent voiceprints, faceprints, etc.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| getSecret | getSecret():String | Returns the print of the biometric |

| | | |
|---|---|---|
| getBiometricIndex | getBiometricIndex():String | Returns the biometric type |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| getSecret | secret:String | Returns the print of the biometric |
| getBiometricIndex | biometricIndex:String | Returns the biometric type |
| username | Username:String | The user associated with the Biometric |

## CheckAccess Class

The CheckAccess class is an implementation of the Visitor class and checks whether or not an entity has been stored within the Authentication Service has matching credentials.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| visit | visit(user:User):boolean | Checks if user in the authentication service has good credentials |
| visit | visit(resource:Resource):boolean | Checks if resource in the authentication service has good credentials |
| visit | visit(role:Role):boolean | Checks if role in the authentication service has good credentials |
| visit | visit(resourceRole:ResourceRole):boolean | Checks if resource role in the |

| | | |
|---|---|---|
| | | authentication service has good credentials |
| visit | visit(permission:Permission):boolean | Checks if permission in the authentication service has good credentials |

## CheckInventory Class

The CheckInventory class is an implementation of the Visitor class and checks whether or not an entity has been stored within the Authentication Service.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| visit | visit(user:User):boolean | Checks if user exists in the authentication service |
| visit | visit(resource:Resource):boolean | Checks if resource exists in the authentication service |
| visit | visit(role:Role):boolean | Checks if role exists in the authentication service |
| visit | visit(resourceRole:ResourceRole):boolean | Checks if resource role exists in the authentication service |
| visit | visit(permission:Permission):boolean | Checks if permission exists in the authentication service |

## Credentials Class

The Credentials class stores the password and biometric information of a given User.

**Methods**

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| add | add(password:Password):void | Adds a password pair to the Credential |
| add | add(biometric:Biometric):void | Adds a biometric pair to the Credential |

## InvalidAuthTokenException Class

The InvalidAuthTokenException is thrown whenever a token has expired or been logged out of, and someone is still trying to use it.

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| getAction | getAction():String | This returns the action performed that caused the exception |
| getReason | getReason():String | This returns the reason why this action was a problem |

**Properties**

| **Property Name** | **Signature** | **Description** |
|---|---|---|
| action | action:String | The action that was performed |
| reason | Reason:String | The reason for the exception |

## Password Class

The Password class represents a credential that holds a given user's hashed ID and password.

**Properties**

| **Property Name** | **Signature** | **Description** |
|---|---|---|

| | | |
|---|---|---|
| username | username:String | The hashed user connected to the password |
| secret | Secret:String | The secret used as the user’s hashed password |

## Permission Class

The Permission class represents singular actions that a particular user or resource can perform.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| accept | accept(visitor:Visitor):boolean | Accepts an object from the visitor class |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| id | id:String | The id of the permission |
| name | name:String | The name of the permission |
| description | description:String | A description of the permission |

## Resource Class

The Resource class represents cities and devices that a particular user or resource can act upon.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| accept | accept(visitor:Visitor):boolean | Accepts an object from the visitor class |

**Properties**

| Property Name | Signature | Description |
|---|---|---|

| | | |
|---|---|---|
| id | id:String | The id of the resource |
| description | description:String | A description of the resource |

## ResourceRole Class

The ResourceRole is a subclass of Role that is attached to a Resource object.

## Role Class

The Role class is a composite collection of roles, resource roles, and permissions. They represent a collection of actions that a user or resource can perform.

**Methods**

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| add | add(role:Role):void | Adds another role within this role |
| add | add(permission:Permission):void | Adds a permission within this role |

**Properties**

| **Property Name** | **Signature** | **Description** |
|---|---|---|
| id | id:String | The role ID |
| name | name:String | The name of the role |
| description | description:String | A description of the role |

**Associations**

| **Association Name** | **Signature** | **Description** |
|---|---|---|
| subroles | subroles:HashMap<String,Role> | A collection of roles and resource roles |
| permissions | permissions:HashMap<String,Permission> | A collection of permissions |

## State Enum

The State class is an enumeration that is either ACTIVE or EXPIRED. This represents the state of an AuthToken.

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| ACTIVE | ACTIVE:State | An AuthToken is ACTIVE when it is usable |
| EXPIRED | EXPIRED:State | An AuthToken is EXPIRED when it is not |

## User Class

The User class represents an inhabitant within the Model Service. The hold their access information such as their usernames, passwords, and biometric prints. Additionally, users can be given access through roles and permissions to perform tasks.

**Methods**

| Method Name | Signature | Description |
|---|---|---|
| addCredential | addCredential(password:Password):void | Sets up a username/password combination for the user |
| addCredential | addCredential(biometric:Biometric):void | Adds a username/biometric combination for the user |
| add | add(role:Role):void | Adds a role to the user |
| accept | accept(visitor:Visitor):boolean | Accepts an object from the visitor class |

**Properties**

| Property Name | Signature | Description |
|---|---|---|
| id | id:String | The User ID |
| name | name:String | The name of the user |
| Credentials | credentials:Credentials | The credentials of the user |

**Associations**

| Association Name | Signature | Description |
|---|---|---|

| | | |
|---|---|---|
| roles | role:HashMap<String,Role> | The roles of the user |
| permissions | permissions:HashMap<String,Permission> | The permissions of the user |

## Visitable Interface

The Visitable interface allows objects to utilize Visitor objects to perform tasks for them.

**Methods**

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| accept | accept(visitor:Visitor):boolean | Accepts an object from the visitor class |

## Visitor Class

The Visitor class allows actions to be performed on objects that implement the Visitable interface

**Methods**

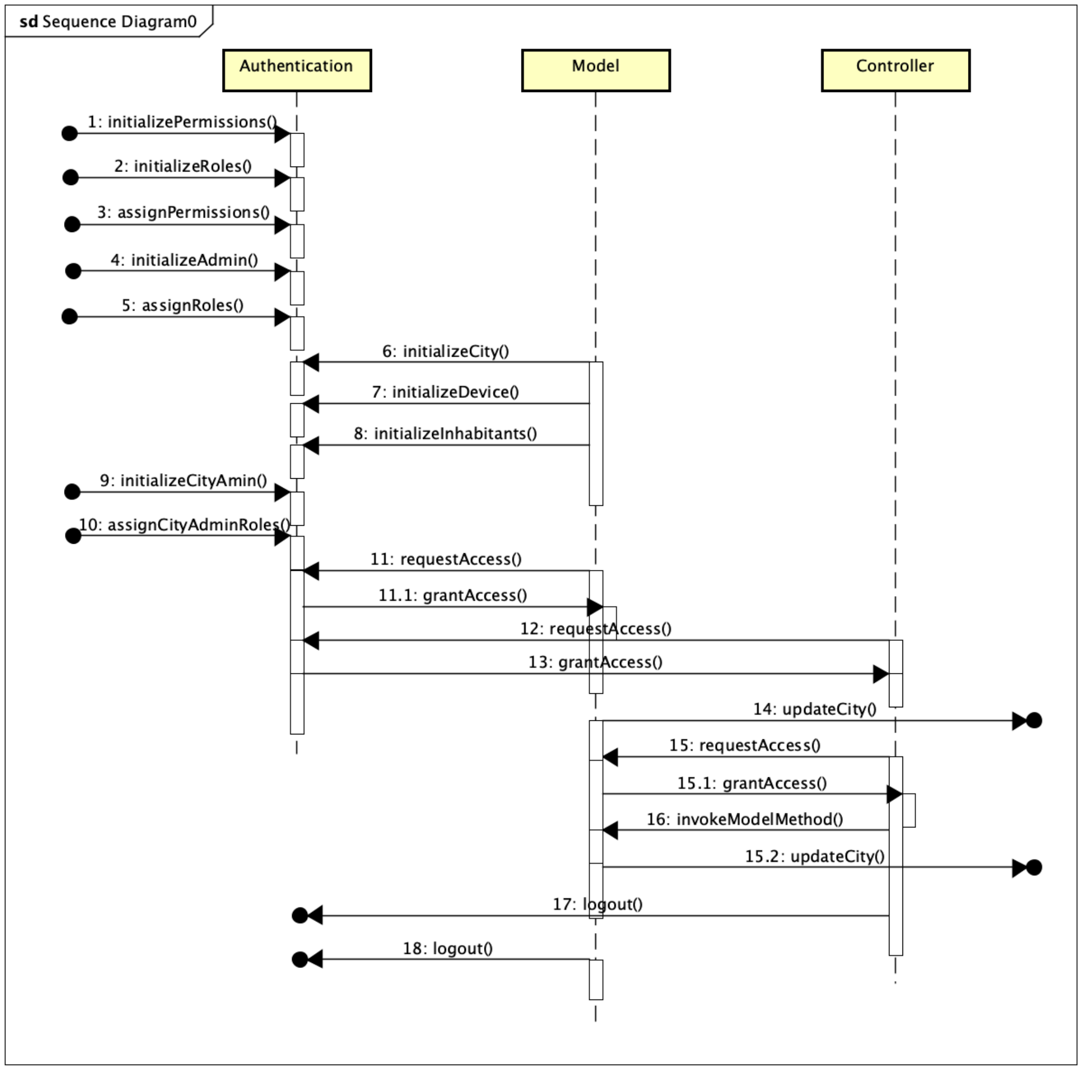
| **Method Name** | **Signature** | **Description** |
|---|---|---|
| visit | visit(user:User):boolean | Allow actions to users |
| visit | visit(resource:Resource):boolean | Allow actions to resources |
| visit | visit(role:Role):boolean | Allow actions to role |
| visit | visit(resourceRole:ResourceRole):boolean | Allow actions to resource roles |
| visit | visit(permission:Permission):boolean | Allow actions to permissions |

## Implementation Details

Below, we have Sequence diagrams that describe the general flow between the Authentication, Controller, and Model Service.

The Authentication service utilizes 3 separate design patterns, namely the Visitor design pattern, the Composite design pattern, and lastly, the Singleton design pattern. The singleton method is applied to the Authentication Service itself, such that all dependents can access the same instance of it. The Visitor method allows us to include methods as we need to that would apply to the entire service without having to modify ever method in every class. The Composite method is utilized through the Roles, Resource Roles, and Permissions.

Upon establishment of the Roles, Permissions, and Administrators, the cities, devices, and even inhabitants can be managed through the 3 services. Upon creation of the primary administrator, a city administrator can then be created, and eventually provided access to perform activities with respect to the city. Similarly, a registered user (or the controller service administrator) can also request and utilize the Authentication service to perform activities as well. Any of these users can have varying degrees of responsibilities dictated by the roles and permissions the are given. Ultimately, if the Authentication service is not aware of them, they won’t have access at all.



## Exception Handling

In addition to the exceptions brought on by the Controller, Model, and Ledger service, the primary exceptions considered in this service are the Authentication, Access Denied, and Invalid Access Exceptions. The Access Denied Exception throws whenever credentials either do not match what is in the Authentication Service, or do not exist at all. The Invalid Access Exception throws whenever an Access Token expires or is currently inactive. Lastly, all other errors, such as reference issues, will throw an Authentication Exception.

## Testing

Below, we have a list of testing strategies required and provided by this service:

- Functional Test
  - Given a pre-defined list off all commands and assure that they function properly and are handled appropriately.
- Performance
  - Assure that there are no hanging processes within the service and that no process runs any longer than is necessary.
- Regression
  - Assure that the inclusion of new functionalities and features do not ultimately bring about errors that did not exist before.
- Exception Handling
  - Capture the exceptions defined above, and return this in a human friendly, but not overly revealing, context.

## Risks

The risks that I identified in the process of this design, were:

- There can be circular Role references if not checked.
- Some roles/permissions can allow multiple users to have an unnecessary amount of access to the city methods.