CSCIE97 Software Design

# House Mate Entitlement Service Design Document

Date: 2017 November 14

Author: Jeremy Clark

Reviewer(s): Ramona Harrison

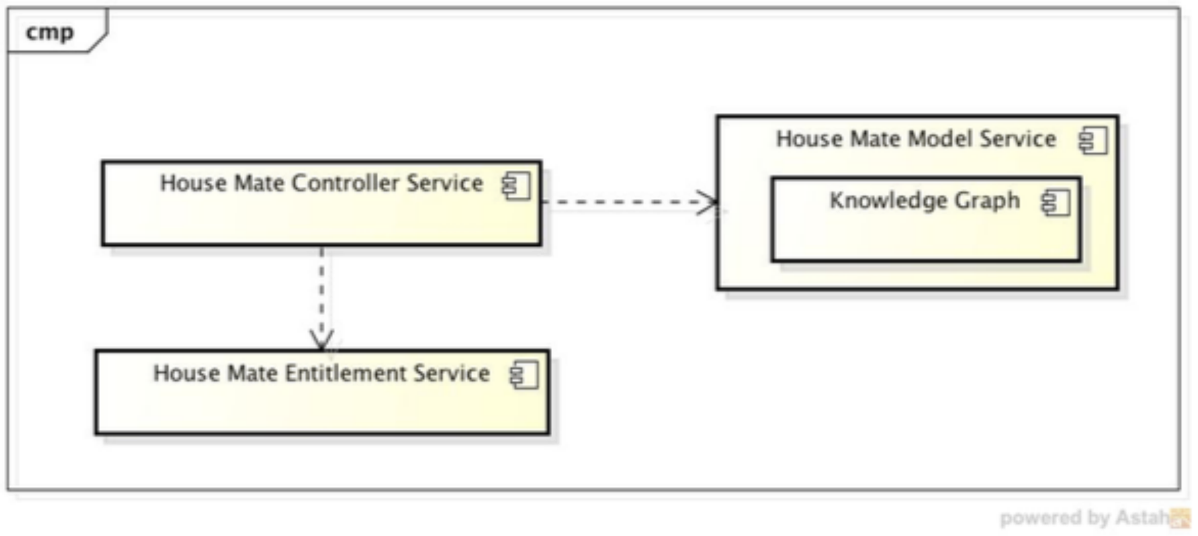
## Introduction

This document defines the House Mate Entitlement Service design.

## Overview

The House Mate home automation system is a comprehensive home automation solution; it supports multiple users with different roles in the home, allows for custom home layouts to support any living space and interfaces with myriad smart devices. The system is highly configurable, and is primarily controlled with voice commands from the home's occupants.

Here's a picture of the overall system architecture:



The subject of this document, the House Mate Entitlement Service ensures secure operation of the system by offering authentication and authorization mechanisms. Each occupant of a house is a user of the system, but different occupants will likely need different access to controls. For example, perhaps only Adults should be able to open the windows and maybe children should only be able to turn operate the TV or Gaming systems during certain windows or when allowed by an Adult. Guests should likely have different access as well, with limited access to private features like certain bedroom access or playback of surveillance videos.

Additionally, system administrators will need access to configuration features (adding/ removing users, roles and permissions) as well as potentially elevated access to other parts of the system.

These different levels of access will be represented by “roles”, which bundle common permission sets into a labeled entity. Each user of the system can be assigned roles which grant them their authorizations for access.

In order to enforce these different levels of access, users of the system must authenticate their identity so the appropriate roles can be applied and permissions granted.

The House Mate Entitlement service will use the following model constructs to achieve its required functionality:

- Users - largely, but not exclusively, map to occupants)
- Resources - protected parts of the system
- Permissions - access grants to specific resources
- Roles - Name sets of permissions that can be applied to users

The creation and configuration of these of these model entities, along with the technical mechanisms to make them work will constitute the entitlement service.

The following document will detail:

- The technical requirements of the system
- A use case summary
- An implementation overview including,

    - Class and sequence diagrams
    - A class dictionary
    - Exception handling considerations
    - An implementation description
- An outline for testing the system
- Outline of risks

## Requirements

Generally, the requirements of the House Mate Entitlement service serve to protect the Privacy and Integrity of the House Mate system. Privacy means that no one who is unauthorized can eavesdrop or otherwise derive what's happening with the system and Integrity means that no unauthorized commands can make their way into the system. Access controls must be provided to ensure no users can inadvertently or maliciously abuse the system or perform unauthorized activity.

More specifically, the requirements of the system dictate:

- As the system is inherently 'closed' to begin, the Entitlement service must allow for the creation of an initial 'administrative' user who can configure and make the system available.
- Administrative users are configured with a username/password combination to protect administrative access.
- Administrative passwords are persisted as hashes so if the user database is compromised, the literal passwords can't be deciphered.
- Administrators can login with their username and password and retrieve an access token to perform configuration tasks like adding users, houses, rooms, devices, etc. to the system via the HouseMateModelService command API.
- Administrators can configure an access token timeout; if the access-token isn't used within that time threshold it will expire and become invalid.

- Access tokens will also have a TTL (time-to-live), after which time it will expire regardless of whether its been recently used.
- If login fails due to an invalid user/pass combo, the user is notified. The notification does not specifically state whether the username or password (or both) were incorrect; it simply states that the user/pass combo was not recognized.
- Administrative users can logout, which invalidates their current access token.
- Administrative users can create other administrative users
- All commands in the Model and Controller service APIs must validate the access token to determine appropriate authorization for execution of the operation.
- Administrative users can create create Resources which represent access-controlled features of then system, like device-features or a ‘type’ of device. The Resources have a description and a UniqueId.
- Administrative users can create Permissions which grant access to resources. The Permissions have a description and can be uniquely IDd in the system.
- Administrative users can create Roles which are bundles of permissions. The Roles provide a construct for aggregating common, reusable sets of permissions that can be applied to Users.
- A Role can be composed of other Roles, individual Permissions or a combination of both.
- A Role can be associated with 0 or more Users.
- Administrators can create Occupants in the HMCS, which will create a corresponding User in the Entitlement Service.
- A User can be assigned 0 or more Roles.
- A Role can be removed from a User (i.e., a user can be disassociated with a Role).
- Occupants access the system through voice commands issued to enabled smart speaker devices.
- Users in the Entitlement service are associated with ‘voice-prints’. When occupants issue voice commands, if the the voice-print is matches a registered Users’ voice-

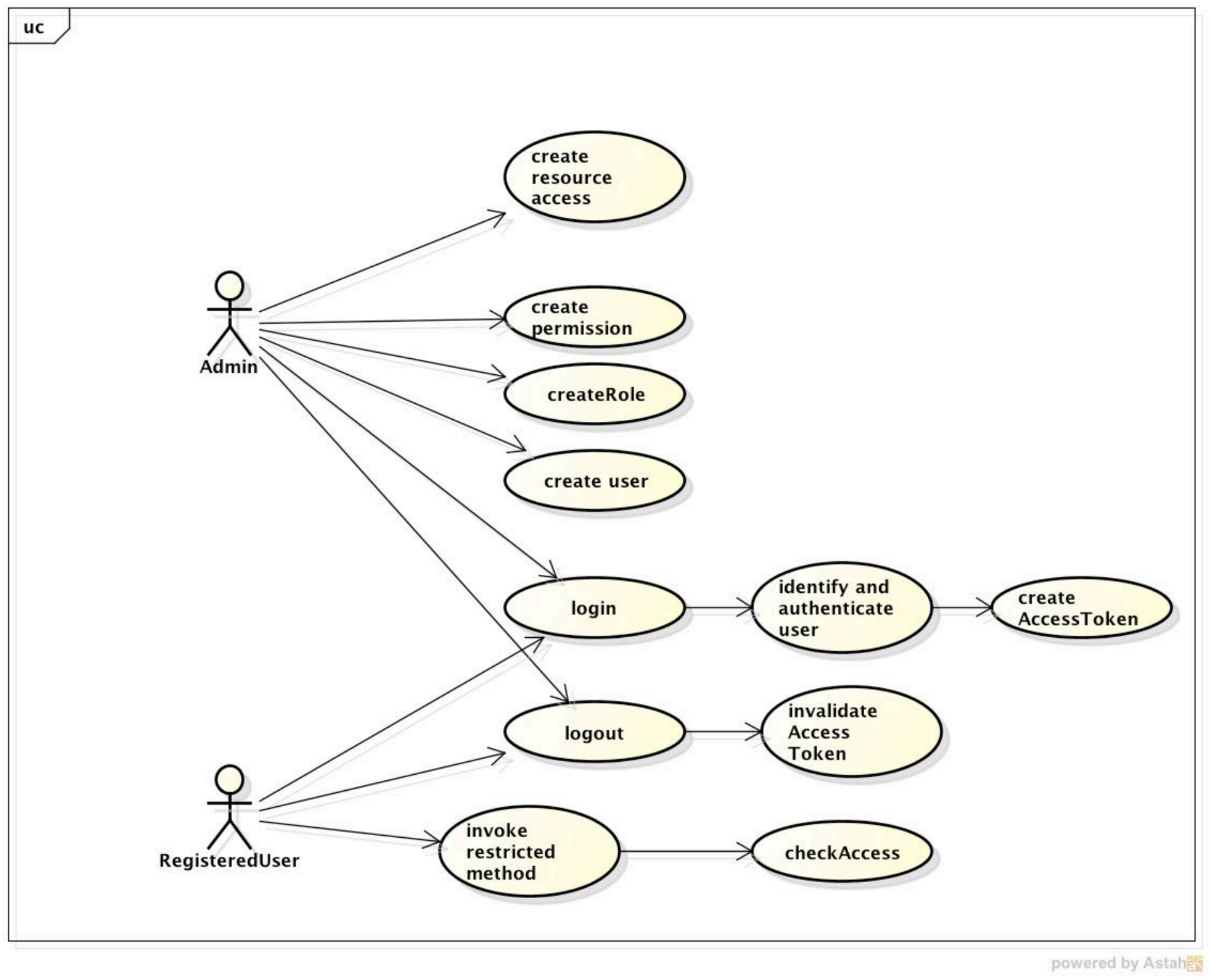
print, that credential is used to retrieve an access token. Like the administrative access tokens, the access token will be used to validate the Users available permissions (inherited through the associated roles) and authorized the requested command.

- If an unauthorized voice command is issued, the user will be notified via a voice response that the request is unauthorized.
- When an occupant is associated with a house, a corresponding Entitlement object will be created that binds the occupant to her role in the house.
- An API must be provided to perform the administrative tasks noted above, which will be available as a Java interface as well as a Command-Line interface.

# Use Cases

## Diagram

The use case diagram provided in the House Mate Entitlement Service requirements document is comprehensive and so is included here for reference. Following the diagram is a detailed description of each use cases.



## Detail

### Create Admin User

*Preconditions*: The person interfacing with the command line has ‘outside-authority’ (In its simplest form, and for the purposes of this document, this means the user has access to the system and the CLI. In a more sophisticated implementation, this may require root-level access to some system resource in order to prove super-user access to some host operating system.)

- The super-user invokes the entitlement service API to create an admin user, including a username and password.
- This supports the ‘bootstrapping’ of the system so an admin user can configure it.

### Administrative Login

*Preconditions:* An admin user exists.

- Admin user passes in a known user/combination password. An access token is returned which grants authority to perform subsequent operations in the system
- Extensions: A ‘bad’ username/combo is passed in, which will return an InvalidCredentialException.

### Administrative Logout

*Preconditions:* An admin user exists, and is logged in

- Admin user invokes the logout command, which will invalidate the currently available access token. Subsequent attempts to the invalidated access token will throw an InvalidAccessTokenException.

### Create Resource

*Preconditions:* An admin user is logged in, and a ‘protected’ entity exists in the House Mate Model

- Admin passes in a reference to a functionality that needs to have controlled access, along with a description of the resource. The resource can refer to any valid FQN from the Model whose root is a “House” identifier. (See HouseMateModelService

Design Document here: *https://canvas.harvard.edu/files/4897580/download?download_frd=1*) This includes:

   - A House
   - A Room
   - A Device
   - A Device Type
   - A Feature

- Use case supports ability to apply granular permissions to functionalities of the system. If a user should have access to all features in their own bedroom, but not the master bedroom, this can be done by creating Room level resources. This is one way that ‘Partial’ access can be granted (for example, a Room is part of a House).

---

## Create Permission

*Preconditions:* Admin user logged in, Resources configured

- Admin creates an access configuration for a specific resource. For example, ‘full-control of the living room door’ or ‘full control of all doors’ or ‘volume control of TV’ or ‘full control of everything in house 1’. In essence this allows for granular access control of every feature in the system, and the ability to assign these controls to various roles.
- Permissions are another mechanism for allowing partial control of a resource. For example, a permission of “volume control of TV” allows for partial control of the TV resource.
- Variations:
   - Create Read Permission: Grants access to the “show” functionality of a resource
   - Create Write Permission: Grants access to the “set” functionality of a resource

- Create Full Permission: Inclusive of both Read and Write permission

---

## Create Role

*Preconditions:* Admin user logged in.

- Admin creates a named role. The Role is used to contain permissions. Additionally, a Role can contain other Roles, to create composite Roles. A Role can then be ‘assigned’ to a User, which will grant that user the permissions associated with the Role (i.e., and permissions granted through the entire composite tree of roles and permissions).

---

## Add Permission to Role

*Preconditions:* Admin user logged in; Roles and Permissions exist.

- Associates a Permission with a Role. By doing this, any Users currently associated with that Role will inherit that permission.

---

## Remove Permission from Role

*Preconditions:* Admin user logged in; Roles and Permissions exist.

- Diassociates a Permission with a Role. By doing this, any Users currently associated with that Role will lose access to that permission. (Unless, they are associated with another role that has that Permission, in which case they will retain access).

---

## Create User

*Preconditions:* Admin user logged in.

- The Admin user doesn’t explicitly create entitlement “Users”, rather, when an Admin explicitly creates a new Occupant in the Model Service, an entitlement service “User” is created.

- A default voice print is created for the User upon creation.

---

## Invoke User voice print update

*Preconditions:* Admin logged in, at least 1 occupant is registered

- The admin user invokes a ‘voice-print’ update in conjunction with an occupant who is physically available.
- Once invoked, the Occupant will have an opportunity to create a new voice print at which point the administrator can accept it and store it with the user.

---

## Create Role-Level Access

*Preconditions:* An admin is logged in; Users, Resources, Permissions and Roles exist.

- This use case allows for associating a User with a Role, and by extension Permissions and their associated Resources. By associating a User with a Role, the administrator ‘grants’ access to the Roles’ Permissions, which authorize the User to perform certain actions in the system.

---

## Revoke Role-Level Access

*Preconditions:* An admin is logged in; Users, Resources, Permissions and Roles exist.

- This use case allows for disassociating a User with a Role, and by extension Permissions and their associated Resources. By disassociating a User from a Role, the administrator ‘revokes’ access to the Roles’ Permissions, which de-authorize the User from performing certain actions in the system.

---

## Invoke Voice Command (Restricted Method)

*Preconditions:* An Occupnant and User exist in a house with a SmartSpeaker (i.e. Ava).

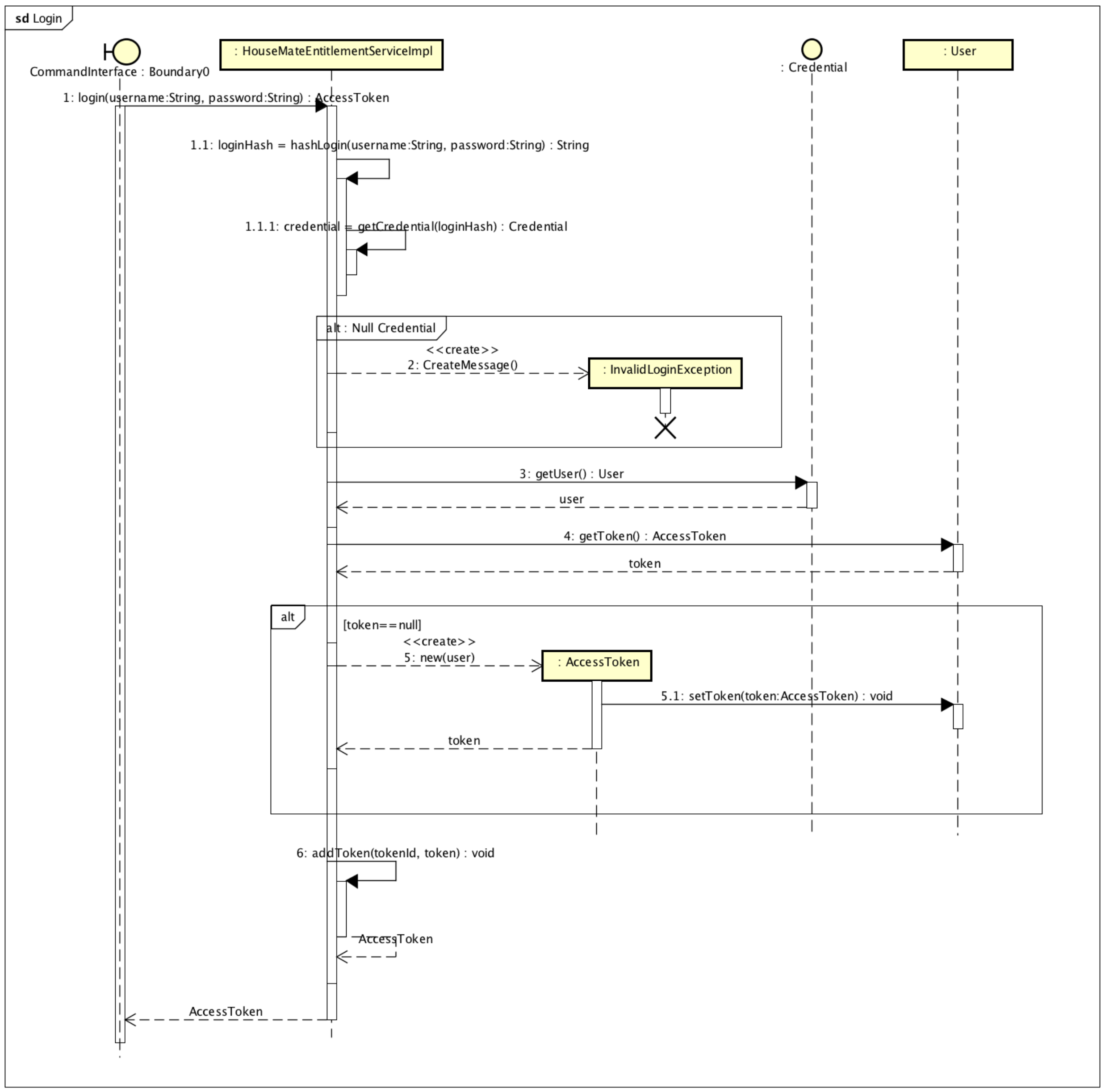
- All voice commands invoke methods which are by default ‘Restricted’. That is all voice-commands will be validated by the Entitlement Service before they are executed. When a voice command is issued, a voice-print will be included - This will invoke the **CheckAccess** use case; if the voice_print credential is found an access token will be generated and passed to the requested command. If the access-token has permission to perform that command, the command will be executed.
- If the generated access token is disallowed by the command, an “UnauthorizedAccesException” will be thrown.
- Extension/Alternative: It is possible to pass an ‘unknown’ voice-print. An unknown voice print will map to an “Unknown” user and “Unknown” role, which may have

some Permissions associated with it. As such, an unknown voice-print may be able to perform some actions in the system.

## Sequence Diagrams

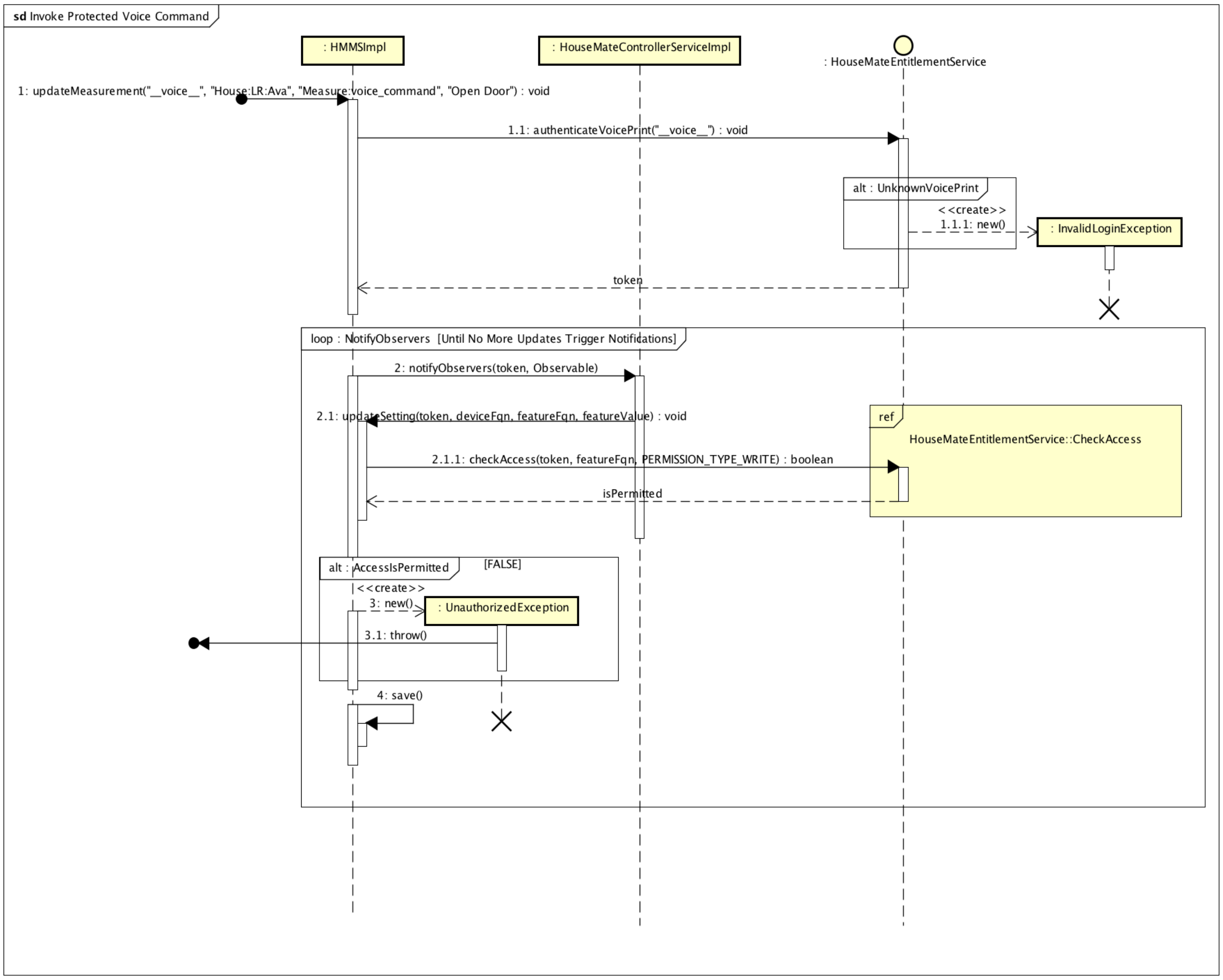
### Login

Illustrates the process for passing in a user/password credential in exchange for a token. If the credential is valid, and there is already a token for the associated user the existing token will be returned. Illustrates the **requirements** above in terms of exchanging credentials for tokens.


sd Login
CommandInterface : Boundary0
: HouseMateEntitlementServiceImpl
: Credential
: User
1: login(username:String, password:String) : AccessToken
1.1: loginHash = hashLogin(username:String, password:String) : String
1.1.1: credential = getCredential(loginHash) : Credential
alt : Null Credential
<<create>>
2: CreateMessage()
: InvalidLoginException
3: getUser() : User
user
4: getToken() : AccessToken
token
alt
[token==null]
<<create>>
5: new(user)
: AccessToken
5.1: setToken(token:AccessToken) : void
token
6: addToken(tokenId, token) : void
AccessToken
AccessToken

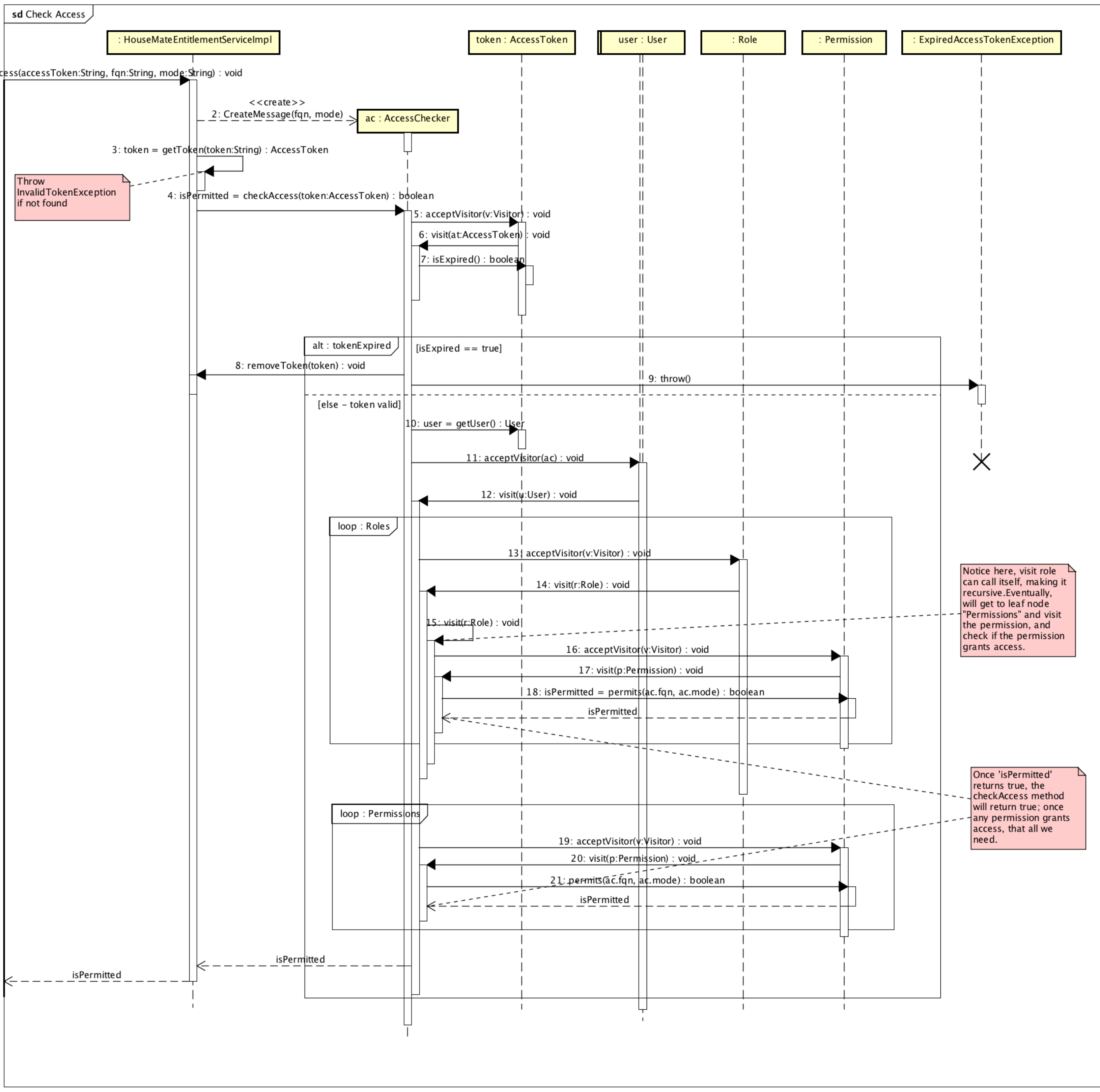
## Invoke Voice Command

Illustrates the process of getting a token for a passed in voice command + voice print, then passing that token to the HouseMateControllerService to authorize model updates associated with the voice command. This illustrates support for the **requirements** involving the execution of model updates via authenticated voice commands.  See the CheckAccess sequence diagram for the details referred to in the HouseMateEntitlementService::CheckAccess reference.


sd Invoke Protected Voice Command
: HMMSImpl
: HouseMateControllerServiceImpl
: HouseMateEntitlementService
1: updateMeasurement("__voice__", "House:LR:Ava", "Measure:voice_command", "Open Door") : void
1.1: authenticateVoicePrint("__voice__") : void
alt : UnknownVoicePrint
<<create>>
1.1.1: new()
: InvalidLoginException
token
loop : NotifyObservers [Until No More Updates Trigger Notifications]
2: notifyObservers(token, Observable)
2.1: updateSetting(token, deviceFqn, featureFqn, featureValue) : void
ref
HouseMateEntitlementService::CheckAccess
2.1.1: checkAccess(token, featureFqn, PERMISSION_TYPE_WRITE) : boolean
isPermitted
alt : AccessIsPermitted
[FALSE]
<<create>>
3: new()
: UnauthorizedException
3.1: throw()
4: save()

## Check Access

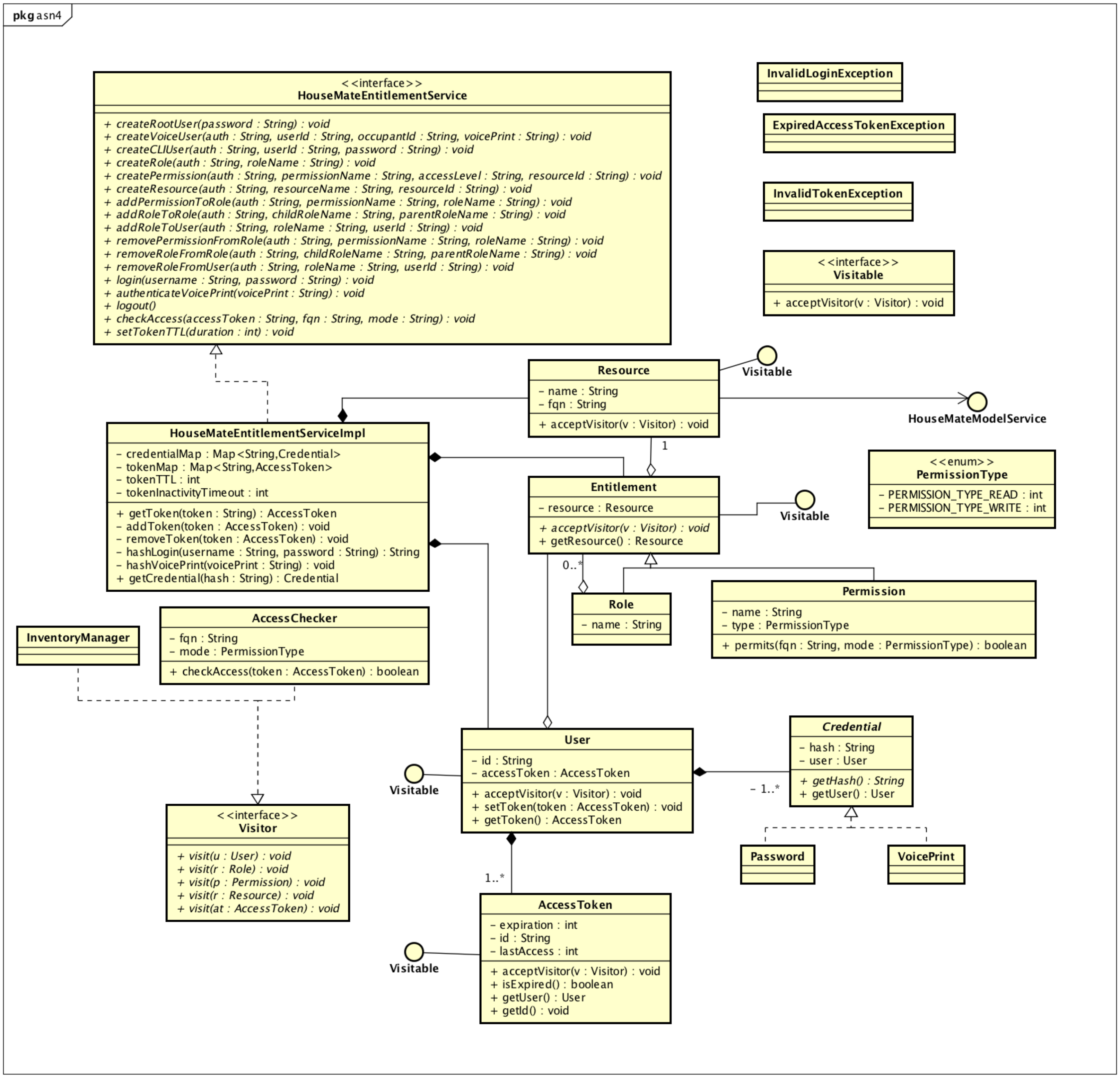
Illustrates the process of checking whether a passed in token is authorized to perform an associated action. Demonstrates use of the visitor pattern to walk the tree of User, Roles and Permissions to find whether the token is valid to perform requested action.


sd Check Access
: HouseMateEntitlementServiceImpl
token : AccessToken
user : User
: Role
: Permission
: ExpiredAccessTokenException
cess(accessToken:String, fqn:String, mode:String) : void
<<create>>
2: CreateMessage(fqn, mode)
ac : AccessChecker
3: token = getToken(token:String) : AccessToken
Throw InvalidTokenException if not found
4: isPermitted = checkAccess(token:AccessToken) : boolean
5: acceptVisitor(v:Visitor) : void
6: visit(at:AccessToken) : void
7: isExpired() : boolean
alt : tokenExpired
[isExpired == true]
8: removeToken(token) : void
9: throw()
[else - token valid]
10: user = getUser() : User
11: acceptVisitor(ac) : void
12: visit(u:User) : void
loop : Roles
13: acceptVisitor(v:Visitor) : void
14: visit(r:Role) : void
15: visit(r:Role) : void
Notice here, visit role can call itself, making it recursive.Eventually, will get to leaf node "Permissions" and visit the permission, and check if the permission grants access.
16: acceptVisitor(v:Visitor) : void
17: visit(p:Permission) : void
18: isPermitted = permits(ac.fqn, ac.mode) : boolean
isPermitted
Once 'isPermitted' returns true, the checkAccess method will return true; once any permission grants access, that all we need.
loop : Permissions
19: acceptVisitor(v:Visitor) : void
20: visit(p:Permission) : void
21: permits(ac.fqn, ac.mode) : boolean
isPermitted
isPermitted
isPermitted

## Implementation

The following section will include a Class Diagram and Class Dictionary followed by some notes describing special parts of the implementation plan.

## Class Diagram


pkg asn4
<<interface>> HouseMateEntitlementService
+ createRootUser(password : String) : void
+ createVoiceUser(auth : String, userId : String, occupantId : String, voicePrint : String) : void
+ createCLIUser(auth : String, userId : String, password : String) : void
+ createRole(auth : String, roleName : String) : void
+ createPermission(auth : String, permissionName : String, accessLevel : String, resourceId : String) : void
+ createResource(auth : String, resourceName : String, resourceId : String) : void
+ addPermissionToRole(auth : String, permissionName : String, roleName : String) : void
+ addRoleToRole(auth : String, childRoleName : String, parentRoleName : String) : void
+ addRoleToUser(auth : String, roleName : String, userId : String) : void
+ removePermissionFromRole(auth : String, permissionName : String, roleName : String) : void
+ removeRoleFromRole(auth : String, childRoleName : String, parentRoleName : String) : void
+ removeRoleFromUser(auth : String, roleName : String, userId : String) : void
+ login(username : String, password : String) : void
+ authenticateVoicePrint(voicePrint : String) : void
+ logout()
+ checkAccess(accessToken : String, fqn : String, mode : String) : void
+ setTokenTTL(duration : int) : void
InvalidLoginException
ExpiredAccessTokenException
InvalidTokenException
<<interface>> Visitable
+ acceptVisitor(v : Visitor) : void
Resource
- name : String
- fqn : String
+ acceptVisitor(v : Visitor) : void
Visitable
HouseMateModelService
HouseMateEntitlementServiceImpl
- credentialMap : Map<String,Credential>
- tokenMap : Map<String,AccessToken>
- tokenTTL : int
- tokenInactivityTimeout : int
+ getToken(token : String) : AccessToken
- addToken(token : AccessToken) : void
- removeToken(token : AccessToken) : void
- hashLogin(username : String, password : String) : String
- hashVoicePrint(voicePrint : String) : void
+ getCredential(hash : String) : Credential
1
Entitlement
- resource : Resource
+ acceptVisitor(v : Visitor) : void
+ getResource() : Resource
Visitable
<<enum>> PermissionType
- PERMISSION_TYPE_READ : int
- PERMISSION_TYPE_WRITE : int
0..*
Role
- name : String
Permission
- name : String
- type : PermissionType
+ permits(fqn : String, mode : PermissionType) : boolean
InventoryManager
AccessChecker
- fqn : String
- mode : PermissionType
+ checkAccess(token : AccessToken) : boolean
Credential
- hash : String
- user : User
+ getHash() : String
+ getUser() : User
User
- id : String
- accessToken : AccessToken
+ acceptVisitor(v : Visitor) : void
+ setToken(token : AccessToken) : void
+ getToken() : AccessToken
Visitable
- 1..*
Password
VoicePrint
<<interface>> Visitor
+ visit(u : User) : void
+ visit(r : Role) : void
+ visit(p : Permission) : void
+ visit(r : Resource) : void
+ visit(at : AccessToken) : void
1..*
AccessToken
- expiration : int
- id : String
- lastAccess : int
+ acceptVisitor(v : Visitor) : void
+ isExpired() : boolean
+ getUser() : User
+ getId() : void
Visitable

# Class Dictionary

## HouseMateEntitlementService

**Interface**

This is the public API that allows for interaction with the EntitlementService.

### Operations

| **Signature** | **Description** |
|---|---|
| public void **createRootUser**(*password*: String) | Creates a root user/password combination that can be used to create administrators. Root user has universal access.<br><br>Takes only a "password" as the static username will be "root" |
| public void **createVoiceUser** (*token*: String, *userId*: String, *occupantId*: String, *voicePrint*: String) | Creates a ‘VoiceUser’, which is maps to a HouseMateModel occupant and uses the passed in voicePrint to create a voicePrint credential. Will employ the **AbstractFactory** pattern to create the appropriate user/ credential combination. |
| public void **createCLIUser** (*auth*: String, *userId*: String, *password*: String) | Creates a “Admin User” that can login from the CLI using a username and password. Technically speaking, users created with this method don’t need to be “Admins” as they can have any roles associated with them like any other users. Will employ the **AbstractFactory** pattern to create the appropriate user/ credential combination. |
| public void **createRole** (*auth*: String, *roleName*: String) | Creates a ‘named role’, which is a composite of other roles and permissions. Roles can be assigned to a user to grant them access to system resource. |

| | |
|---|---|
| public void<br>**createPermission**<br>*auth*: String,<br>*permissionName*: String,<br>*accessLevel*:PERMISSION_TYPE,<br>*resourceId*: String) | Creates a “Permission” that allows access to a Resource object. Resources are mapped to HouseMateModel Config Item FQNs (see HouseMateModel design doc for details), and the accessLevel is either PERMISSION_TYPE_READ (get) or PERMISSION_TYPE_WRITE (set) |
| public void<br>**createResource**<br>(*auth*: String,<br>*resourceName*: String,<br>*resourceId*: String) | Creates a resource, which is basically a wrapper reference to a HouseMateModel ConfigItem FQN. resourceId is any valid FQN and can be a House, Room, Device, Setting or Measure. For example: createResource(“House1 Living Room”, “House1:LR”) will register the entire LR of house1 as a Resource.<br><br>Once this resource exists, Permissions can be created to grant access to that resource. For example: createPermission(“House 1 Living Room Full Access”, PERMISSION_TYPE_WRITE, “House1:LR”) will allow full read/write access to any HMMS ConfigItem descendant of House1:LR. |
| public void<br>**addPermissionToRole**<br>(*auth*: String,<br>*permissionName*: String,<br>*roleName*: String) | Adds a permission to a role, and grants that access to any users already associated with that Role.<br><br>Throws ItemNotFoundException |
| public void<br>**addRoleToRole**<br>(*auth*: String,<br>*childRoleName*: String,<br>*parentRoleName*: String) | Adds a Role as a child of another Role, creating a Composite role. Will check that child is not already a child of parent or any of parent’s ancestors.<br><br>Throws ItemNotFoundException |
| public void<br>**addRoleToUser**<br>(*auth*: String,<br>*roleName*: String,<br>*userId*: String) | Adds a role to a user, which grants all permissions in the Role or any of its descendant Roles.<br><br>Throws ItemNotFoundException |

| | |
|---|---|
| public void<br>**removePermissionFromRole**<br>(*auth*: String,<br>*permissionName*: String,<br>*roleName*: String) | Removes a permission from a Role, which will apply to any users associated with roleName or any of its ancestors.<br><br>Throws ItemNotFoundException |
| public void<br>**removeRoleFromRole**<br>(*auth*: String,<br>*childRoleName*: String,<br>*parentRoleName*: String) | Removes a role from its parent.<br><br>Throws ItemNotFoundException |
| public void<br>**removeRoleFromUser**<br>(*auth*: String,<br>*roleName*: String,<br>*userId*: String) | Removes a role from a user, effectively revoking any Permissions descendant from that Role. |
| public AccessToken<br>**login**<br>(*username*: String,<br>*password*: String) | Checks the passed in username/password against known credentials. If it's valid returns an AccessToken. |
| public AccessToken<br>**authenticateVoicePrint**<br>(*voicePrint*: String) | Checks the passed in voicePrint string against known credentials. If it's valid returns an AccessToken. |
| public void<br>**logout**<br>(*username*: String) | CLI logout command, which will remove the current accessToken for username |
| public boolean<br>**checkAccess**<br>(*accessToken*: String,<br>*fqn*: String,<br>*mode*: String) | Will check wether the accessToken provided authorizes the requested command 'mode' for the resource associated with 'fqn'. This will be the entry point from the Model and Controller Service when checking whether a command is authorized.<br><br>If the the request is unauthorized, and the accessToken is for a voiceUser, this method will issue a command to all Ava devices in the house that the request was unauthorized. This meets the **requirement** that unauthorized voice access will trigger a voice response from Ava.<br><br>Returns true if authorized, false if not. |

| | |
|---|---|
| public void **setTokenTTL** (*duration*: int) | Meets **requirement** of allowing admins to set amount of time before a token expires. |

## HouseMateEntitlementServiceImpl

### Implements <<HouseMateEntitlementService>>

Implementation of the HouseMateEntitlementService. Holds references to Users, Entitlements (Roles and Permissions) and Resources. Also maintains a map of the active AccessTokens and Credentials.

Will create a timer that fires at intervals equal to tokenInactivityTimeout that will invoke the "pruneInactiveTokens" method. This meets the **requirement** of invalidating tokens that have been inactive for a specified period of time.

NOTE ON AUTHENTICATION OF HMES METHODS: In order to authenticate the methods provided by the Interface, this implementation of the public API methods will pass "HMES" as the 'fqn' to the checkAccess method. A Resource must exist with HMES as the FQN, and a Role associated with admin users that grants access to the HMES in order for these methods to be executed. This is how the **requirement** of protecting access to the HouseMateEntitlement service will be met.

**Associations**

| Reference | Description |
|---|---|
| **users** | The registered Users in the EntitlementService |
| **entitlements** | The set of Entitlements (Roles and Permissions) available in the Service |
| **resources** | The registered resources in the Entitlement Service. |

**Attributes**

| Name | Description |
|---|---|
| Map<String, AccessToken> **tokenMap** | A map of the currently active tokens |
| Map<String, Credential> **credentialMap** | A map of currently known credentials, where key is the hashString of the credential. |
| private int **tokenTTL** | Stores the default duration before a token expires. |

| | |
|---|---|
| private int **tokenInactivityTimeout** | Duration after which an inactive token should be invalidated. |

**Operations**

| Signature | Description |
|---|---|
| private AccessToken **getToken** (*tokenString*: String) | Retrieves an AccessToken object from the tokenMap. |
| private AccessToken **addToken** (*token*: AccessToken) | Adds a token to the map. |
| private AccessToken **removeToken** (*token*: AccessToken) | Removes a token from the map. |
| private String **hashLogin** (*username*: String, *password*: String,) | Creates a 'hashed' version of a username/ password combination. Will use basic one-way hashing like MD5. This meets the requirement of storing an undecipherable password. |
| private String **hashVoicePrint** (*voicePrintString*: String) | Creates a 'hashed' version of a voicePrint. Will use basic one-way hashing like MD5. This meets the requirement of storing an undecipherable credential in case of compromise. |
| private Credential **getCredential** (*credentialHash*: String) | Retrieves a Credential object from the credentialMap. |
| private void **addCredential** (*credential*: Credential) | Adds a credential to the map |
| private void **removeCredential** (*credentialHash*: String) | Removes a credential from the map. |
| private void **removeCredential** (*credentialHash*: String) | |

| | |
|---|---|
| private void **pruneInvactiveTokens** () | Will loop through tokens in the token map to see if the 'last activity time' is older than the tokenInactivityTimeout allows for. If the token is has been inactive, this method will remove the token from the map and destroy it. This method will be set to fire on a timer when the HMESImpl is initialized. |

## Visitable

### Interface

A simple interface that provides one method acceptVisitor(v:Visitor).

**Operations**

| Signature | Description |
|---|---|
| protected void **acceptVisitor** (visitor: Visitor) | Accepts an object that implements Visitor. Implementors will provide their own implementations, but they will all basically call visitor.visit(this). |

## Resource

### Implements <<Visitable>>

A protected Resource in the Entitlement service. Maps to a HouseMateModel ConfigItem FQN - (see HouseMateModelService design document here: https://canvas.harvard.edu/files/4897580/download?download_frd=1) . In addition the HouseMateModel config items, callers can also prepend an FQN pass with "HMCS" or "HMES" to represent the ControllerService and EntitlementService resources respectively. This supports the **Requirement** that the HMES and HMCS creational methods are restricted to admin users only. By creating resources for the HMES and HMCS, permissions can be created to grant access to the methods in those services.

**Associations**

| Reference | Description |
|---|---|
| **entitlements** | A Resource can be associated with one or more Entitlements (Roles and Permissions). The main use case is to associate a Resource with a permission, which will grant Read or Write access to that resource (and all of that resource's descendants). |

**Attributes**

| Name | Description |
|---|---|
| String **name** | The name of the Resource for use in the EntitlementService |
| String **fqn** | The HouseMateModel ConfigItem FQN of the resource OR an fqn prepended with HMCS or HMES. |

## User

### Implements <<Visitable>>

User in the Entitlement service. Represents both CLI users (admins) and Voice users (occupants), with the type distinction being handled by the type of associated Credentials. Implements "Visitable" so it can be processed as part of Entitlement Service tree walks processes.

**Associations**

| Reference | Description |
|---|---|
| **credentials** | The credentials registered for the User. |
| **entitlements** | The set of Entitlements (Roles and Permissions) granted to the user |

**Attributes**

| Name | Description |
|---|---|

| | |
|---|---|
| String **id** | The userId. For users that are created as associations to Occupants in the HouseMate model system, this will be the HouseMate occupant username. |
| AccessToken **token** | The currently available AccessToken for the user. Can be null. |

**Operations**

| Signature | Description |
|---|---|
| protected AccessToken **getToken** () | Retrieves the current AccessToken |
| protected void **setToken** (*token*: AccessToken) | Sets the current token |

---

## AccessToken

### Implements <<Visitable>>

Represents an access token that maps to a User. Maintains state about when it expires.

**Attributes**

| Name | Description |
|---|---|
| private String **id** | The accessToken ID. This will be system generated GUID. |
| private int **expiration** | The expiration timestamp. Will be a UNIX timestamp, passed in on creation. For this design, will reflect the **tokenTTL** from the HMES. |
| private int **lastActivityTime** | A timestamp reflecting the last activity. This will be updated by the “isExpired()” method which will be called every time a token is checked for authorization. This state will be used by the HMESImpl::pruneInactiveTokens() method to meet the **requirement** of invalidating inactive tokens. |
| private final User **user** | The user associated with the accessToken |

**Operations**

| Signature | Description |
|---|---|
| protected boolean **isExpired** () | Checks whether the **expiration** is in the past. Will update the lastActivityTime to the time at which this method is invoked.<br><br>True if expired, false if not. |
| protected User **getUser** () | Gets the current User |
| protected String **getId** () | Gets the Token ID String. |

---

## Credential

### Abstract

Abstract class that represents an credential that can be used to authenticate a User. Concrete subclasses will implement the hash methods.

**Attributes**

| Signature | Description |
|---|---|
| private String **hash** | A "hash" value for the credential, which obscures the underlying authentication information (for example, a password). An implementor could theoretically store the authentication information without obscuring it. |
| private User **user** | The user that owns the credential. |

**Operations**

| Signature | Description |
|---|---|
| public **Credential**(credentialString, User) | Initializes a credential by taking a credential string (like a concatenated user/pass, or a voice_print), hashing it with a basic MD5 hash function and storing the hash. |
| protected String **getHash** () | Returns the hash attribute. |

| | |
|---|---|
| protected User **getUser** () | Returns the user associated with the |

## Password

### Extends Credential

Concrete credential subclass that represents a “username/password”.

**Operations**

| Signature | Description |
|---|---|
| public **Password**(credentialString, User) | Will use the default Credential constructor and use MD5 hashing of the passed in string. |

## VoicePrint

### Extends Credential

Concrete credential subclass that represents a “voice_print” credential.

**Operations**

| Signature | Description |
|---|---|
| public **VoicePrint**(credentialString, User) | Will use the default Credential constructor and use MD5 hashing of the passed in string. |

## Entitlement

### Implements <<Visitable>>

Represents an ‘Entitlement’ in the EntitlementService. This is an abstract composite component type used when creating composite Role objects (Roles that contain other Roles as well as Permissions). A tree of entitlements Roles Containing Roles and Permissions is effectively what grants access to a User.

**Associations**

| Reference | Description |
|---|---|

| | |
|---|---|
| **users** | An entitlement is associated with 0 or more Users in order to grant them access to resources. Permissions are really the objects that grant access, but a User may only be associated with Roles that grant access through descendant Permissions. |

**Attributes**

| Reference | Description |
|---|---|
| private final **resource** | An entitlement has an association with one Resource. The associated Resource is used to govern access via the associated Entitlement. |

**Operations**

| Reference | Description |
|---|---|
| private Resource **getResource** () | Gets the resource associated with the Entitlement. |

## Role

### Implements <<Visitable>>, extends Entitlement

Represents an 'Role' in the EntitlementService. This is a composite structure that can contain other Entitlements (both Roles and Permissions), and can be assigned to Users to grant them access.

**Associations**

| Name | Description |
|---|---|
| **entitlements** | A role contains 0 or more Entitlements, which given the fact that a Role is an entitlement means it can contain other Roles. This creates a **composite structure** of Roles and Permissions. |

**Attributes**

| Signature | Description |
|---|---|
| private String **name** | The name of the Role. |

**Operations**

| Signature | Description |
|---|---|
| protected String **getName** () | Get the Role name |

## PermissionType

### Enum

**Values**

| Signature | Description |
|---|---|
| int PERMISSION_TYPE_READ | Grants read access when associated with a Permission. |
| int PERMISSION_TYPE_WRITE | Grants both read and write access when associated with a Permission. |

## Permission

### Implements <<Visitable>>, extends Entitlement

Represents an ‘Permission’ in the EntitlementService. This is the object that specifically allows an action to be executed on behalf of a user. It has a “mode” (read/write) and a reference to a Resource. A user can be associated with a Permission either directly or via a Role that has the Permission as a descendant.

**Associations**

| Name | Description |
|---|---|
| **users** | Described in the association section in the parent class Entitlement. |

**Attributes**

| Signature | Description |
|---|---|
| private String **name** | The name of the Permission. |

| | |
|---|---|
| private PermissionType **type** | The “mode” of the permission. Either PERMISSION_TYPE_READ or PERMISSION_TYPE_WRITE (which is inclusive of PERMISSION_TYPE_READ) |

**Operations**

| Signature | Description |
|---|---|
| protected boolean **permits** (fqn:String, mode: PermissionType) | Checks whether the Permission permits access to perform an operation of type “mode” on the passed in FQN. The **fqn** is matched against the ID of the Permission’s and the **mode** is matched against the Permissions mode attribute. If the **fqn** is equal to or a descendant of this.getResource.getFqn() AND the permisson’s type is inclusive of the passed in **mode**, then this will return “true”. For example, if permission.type == PERMISSION_TYPE_WRITE and permission.resource.getFqn() == “house1:room1”, then permission.permits(“house1:room1:device1”, PERMISSION_TYPE_READ) will return true. |

## <<Visitor>>

### Interface

Visitor interface that provides method signatures to visit all of the visitable types.

**Operations**

| Signature | Description |
|---|---|
| public void **visitRole**(r: Role) | Visits the passed in Role. |
| public void **visitPermission**(p: Permission) | Visits the passed in Permission. |
| public void **visitUser**(u: User) | Visits the passed in User. |
| public void **visitToken**(t: AccessToken) | Visits the passed in Token. |

|  |  |
|---|---|
| public void **visitResource**(r: Resource) | Visits the passed in Resource. |

## AccessChecker

### Implements <<Visitor>>

Visitor class that performs the primary operation in the EntitlementService which is checking access for a requested operation.  When the public HMES checkAccess() api method is called, it will create an AccessChecker by calling new AccessChecker(fqn, mode) object and call AccessChecker::checkAccess(token). The check access method will kickoff a process by calling "acceptVisitor(token)" for the passed in token which will invoke visitToken() and a tree walk will ensue. To see the details of the control flow, refer to the Check Access sequence diagram in earlier in this document.

**Associations**

| Name | Description |
|---|---|
| **users** | Described in the association section in the parent class Entitlement. |

**Attributes**

| Signature | Description |
|---|---|
| private String **fqn** | The fqn of the resource that is being operated on in the HouseMateModel Service. |
| private String **mode** | The "mode" of the requestedAccess. Either PERMISSION_TYPE_READ (when a 'show' method is being invoked in the HMMS) or PERMISSION_TYPE_WRITE (when a 'set' method is being invoked in the HMMS). |
| private boolean **accessGranted** | Local state variable that tracks whether the check access visitor walk has found a permission that grants access. Initialized to false. |

**Operations**

| **Signature** | **Description** |
|---|---|
| public boolean **checkAccess** (token:AccessToken) | The check access method will kickoff a process by calling "acceptVisitor(token)" for the passed in token which will invoke visitToken() and a tree walk will ensue. To see the details of the control flow, refer to the Check Access sequence diagram in earlier in this document and see the "visitX" implementation details below.<br><br>At the end of the visitor dispatches, this method will return the value of "accessGranted", which will have been set to true if any of the visited permissions.permits() method returns true. |
| public void **visitToken** (token:AccessToken) | Will call "isExpired()" on the passed in **token.** If it's expired, will throw an ExpiredTokenException. If it's not expired, will call token.getUser().acceptVisitor(self) |
| public void **visitUser** (user:User) | Will call user.getEntitlements() and loop through all entitlement, calling entitlement.acceptVisitor(self). If the entitlement is a Role, then "visitRole()" will be invoked, if it's a permission then "visitPermission()" will be invoked. |

| | |
|---|---|
| public void **visitRole** (role:Role) | Will call role.getEntitlements() and loop through all entitlements, calling entitlement.acceptVisitor(self). If the entitlement is a Role, then "visitRole()" will be invoked which manifests as a recursive walk over the Role composite structure. If it's a permission then "visitPermission()" will be invoked. |
| public void **visitPermission** (permission:Permission) | Will call permission.permits(**fqn, mode)** where fqn and mode are the local attributes passed in upon construction. If permission.permits() returns true, then the local **accessGranted** value will be set to true. |

## Exceptions

---

### ExpiredAccessTokenException

**Extends Exception**

Thrown by the AccessChecker::checkAccess() method when a provided access token has expired.

---

### InvalidAccessTokenException

**Extends Exception**

Thrown by the AccessChecker::checkAccess() method when a provided access token is not valid (i.e., not found in the tokenMap).

---

### InvalidLoginException

**Extends Exception**

Thrown by the HMES::login() and HMES::authenticateVoicePrint() methods when the passed in credentials are not found in the credentialMap.

## Implementation Details

- Note there is no "ResourceRole" class. I didn't see what state or behavior the Association class would provide beyond the direct association between an entitlement and a Resource.
- The **Visitor** pattern is implemented through use of the AccessChecker visitor which implements methods to visit Tokens, Users, Roles, Permissions and Resources. As of now, there is not "Visit Resource" implementation specified, as its not required to validate access in this design.
- The **Composite** pattern is used to provide complex "Roles" which can contain other roles.
- The **AbstractFactory** pattern will be used by the HMESImpl to create the appropriate Credential objects in the createRootUser, createVoiceUser and createCLIUser implementations. I guess I could generalize those public API methods to a single 'createUser' method that takes a 'type' parameter; then pass that to an abstract UserFactory which will decide what type of user to create. If I do that, however, I'll need to change the parameter set to take 'no password' for the root user , a user/pass combo for the admin/cli user and a voice_print for the voice users. This could be achieved by taking a single string - which will work for all but the CLI user, though callers could concatenate the user/pass with a known delimiter like a ":" or something else (or possibly no delimeter).
- The **Singleton** pattern will be used to get access to the HMES impl, as well as the HMCS and HMMS implementations as necessary.
- The current design will always walk the entire tree of Entitlements. It might be
- At a high level, the sequence diagrams, class diagram and class dictionary offer details about how the design meets the requirements. Some additional notes worth mention here:
  - The HMES api provides interfaces to setup and configure entitlement objects.
  - The createRootUser command allows for creating the "intial" user than can then create more admins.

- The createVoiceUser command can be invoked by the HMMS when an occupant is created in order to register voice_prints for HMMS occupants.

## Testing

The high-level testing approach is as follows:

- Extend the command interface to include entitlement service configuration options.
- Extend the command interface for HMMS to include accepting a “voice_print” parameter
- Use the existing controller setup scripts included with ASN3 to bootstrap a model and controller configuration.
  - Note: the command to sense a voice command will not be protected.

- Resources, Roles, Permissions and Users will then be created to test the following scenarios:

  - Grant full write control of a house, then attempt to update a child device in the house.

  - Grant full write control of a room, then attempt to:
    - Update a child device in the room
    - Update a child device of the same type in another room

  - Grant full write control of a single device, the attempt to:
    - Update a feature on that device
    - Update a feature on a different device

  - Grant full write control of a single feature, then attempt to:
    - Update that feature
    - Update a different feature on the same device

  - Create a composite Role (role containing another role), assign to user, and attempt to:
    - Perform an operation that exercises a nested permission

  - Create Two permissions. Grant one to a User, then:
    - Perform an operation that exercises the granted permission
    - Perform an operation that exercises the non-granted permission (should be rejected)
    - Grant the second permission to the user and try again (should work)
    - Revoke second permission and try again (should fail)
    - Add second permission to a role and add role to user. Try again (should work).

- Consider a voice command that will create multiple Model updates (through the observer notifications in the controller service.) Setup permissions such that one of the commands will be unauthorized (Should Fail)
  - Add permissions for all commands and try again (Should succeed).

## Risks

- Using MD5 as a hash is insecure, but illustrates the requirement of hashing the credentials (passwords/voiceprints).
- The creation of a 'root' user introduces a single, high-stakes vulnerability.
- Care will need to be taken to avoid closed cycles within the Entitlement graph. Because the API allows for adding a possibly complex Role as a child to another possibly complex Role, I'll need to make sure that the child Role is not a descendant of itself - otherwise the checkAccess recursive visit could infinitely loop.
- If an Entitlement tree is large, it could take a long time to walk the visit the entire tree which could delay authorization of a request. One possible way to mitigate this might be to maintain a flattened collection of all permissions provided at every node in the tree. This could then propagate up to the Roles associated with each User, and the User could then have a managed 'index' of permissions which could be accessed to avoid a tree walk when checking access. The tradeoff of this approach is that each update to the entitlement tree (Adding a permission to a role, adding a role to a role, adding a role to a user) will require a complex tree walk and update in order to update state of the entire graph…which could also take a long time and be complicated.