Authentication Service Design Document

Date: 11/9/2019 (revised 11/20/2019)
Author: Matthew Thomas
Reviewer(s): Eric Gieseke (Ramnath Pillai was not able to provide feedback; James McCrary never responded to forms of contact)

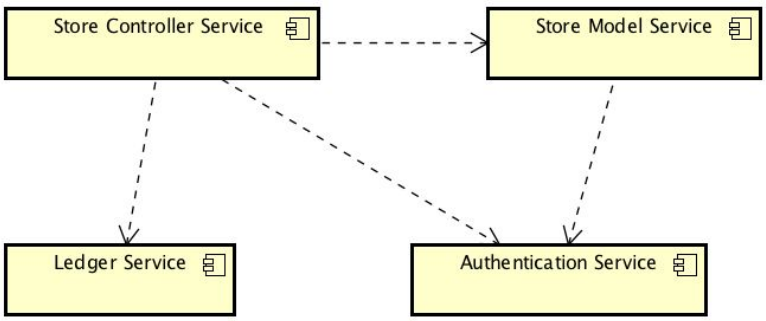
Introduction

This design specifies the implementation of an Authentication Service, one component of the Store 24X7 Software System. The Authentication System is the security behind the system and provides a way to create Entitlements that permit or restrict Users from performing actions.

Overview

As the modules of the Store 24X7 System have been constructed throughout assignments 1-3, a missing piece has been the Authentication Service. Up until now, the Store 24X7 System can define multiple Stores and their respective layouts and Product offerings (Store Model Service), manage the state of the Store through Commands in response to events or Customer queries (Store Controller Service), and integrate with a payment system to process Transactions (Ledger Service).

A requirement that has been lacking throughout these modules is authentication: who can access what. Security is a big concern for an automated store system, especially one that contains numerous Internet of Things (IoT) connected Devices. Hackers can exploit the system by calling the public APIs of the individual services and well-meaning Customers could unwittingly access certain areas that might be restricted (e.g. a Child entering the liquor aisle, or any Customer entering the stock room meant for employees only).



This component diagram illustrates where the Authentication Service comes into play. This assignment focuses on the Authentication Service which is a dependency of the Store Controller and Store Model Services. Details about how they interact can be found in the “Implementation” section below.

Document Organization

This design document provides a list of requirements the Authentication Service will need to address, and use case scenarios about how actors will interact with the service. This document also includes a class diagram and dictionary to assist a programmer with an implementation, and sequence diagrams to demonstrate how interactions between the Authentication, Controller, Ledger, and Store Model Service take place. Furthermore, the design touches upon exception handling, testing, and known risks in the Authentication Service that should be taken into consideration when the system is deployed.

Requirements

This section defines the requirements for the Authentication Service.

Authentication Service

1. Create Entitlements (Roles, Permissions, Resource Roles).
2. Create Resources that refer to physical entity.
3. Create Users to correspond to Customers in a Store and Accounts in the Ledger Service.
4. Validate User credentials.
5. Generate AuthTokens.

Users

1. Register with the Authentication Service.
2. Admin Users can provision the Authentication Service as well as the Store Controller, Model and Ledger Services.
3. All Users can invoke calls to public API methods. If the User does not have the appropriate level of permissions, an Exception will be thrown.

Entitlements (Roles/Permissions)

1. Define groups of permitted behavior for accessing restricted methods.
2. Permissions define what behavior is allowed. Roles and ResourceRoles

Store Model Service

1. Accept an AuthToken for all public API methods. This token is then passed to the Authentication Service to validate that the token is valid and the caller has the appropriate level of permissions.

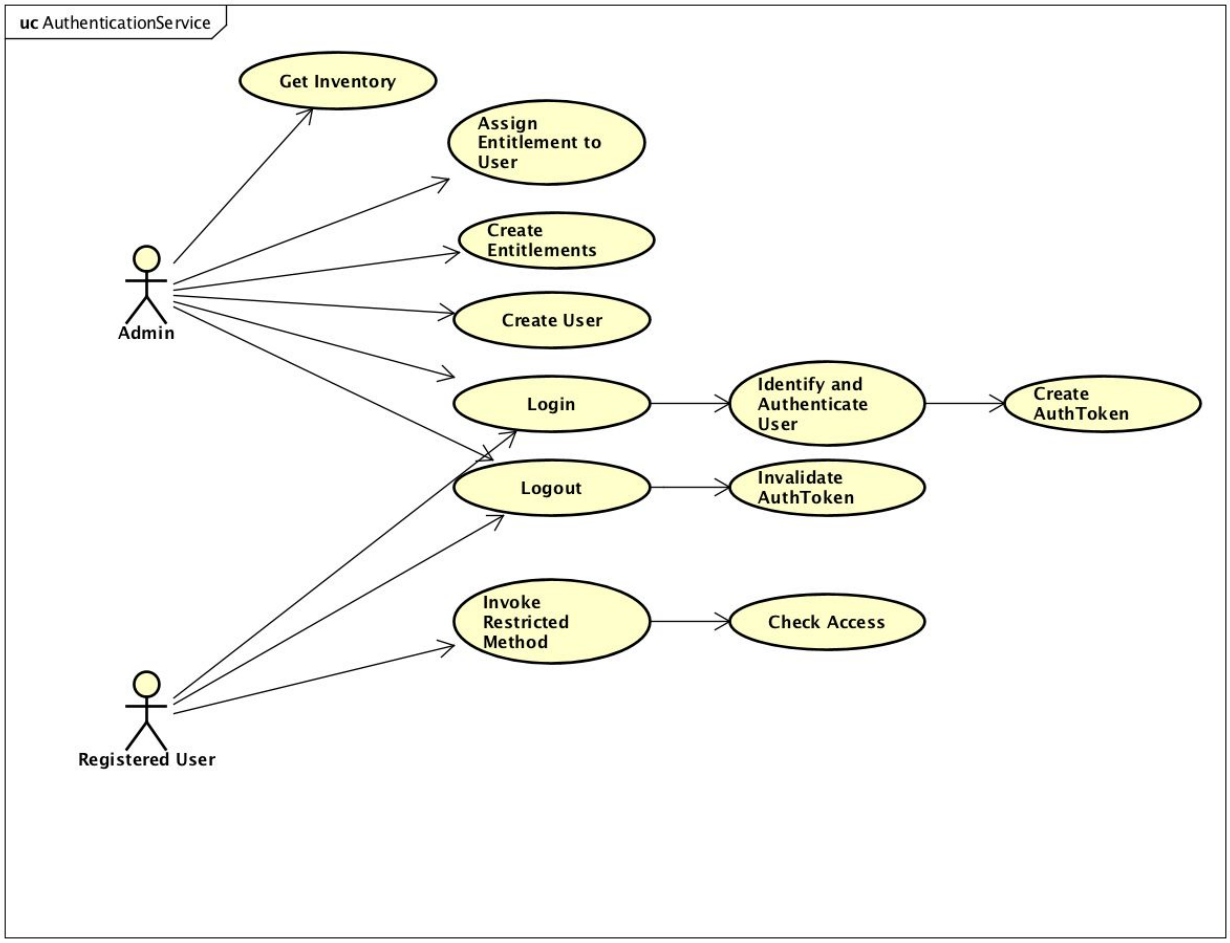
Not required

Persistence

As in assignments 1, 2, and 3, the Authentication Service is maintained in memory. Saving the state of the Authentication Service is not required beyond keeping the objects in memory.

Use Cases

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



Actors

The actors of the Authentication Service include Admins and Registered Users.

Admin

The Admin is responsible for creating the Users, Resources, and Entitlements (Roles, Permissions) that compose the Authentication Service. The Admin assigns Entitlements to Users to give Permissions and restrict access to other methods. The Admin can also login and logout.

Registered User

A Registered User can login or logout of the system when entering or exiting a Store. While in a Store, the User can perform an action (e.g. add item to basket) and succeed or fail depending on the Permissions assigned to that User.

Use Cases

Assign Entitlement to User

To grant a User permission to access a restricted method, they must be assigned an Entitlement (ResourceRole, Role, or Permission). The Authentication Service requirements (page 2) mention that this assignment is done when a Users are linked to Customers in the Store Model Service.

Create Entitlements

Entitlements are ResourceRoles, Roles, and Permissions. These follow the Composite Pattern (*Head First Design*, Chapter 9) where Roles (of which a ResourceRole is a role that is associated with a specific Resource) are composites of Permissions. In other words, Roles can contain other Roles and Permissions whereas Permissions are leaf nodes which represent a permission required to access a given resource (requirements, page 5).

Create User

When a Customer is added to the Store Model Service, a corresponding User is created within the Authentication Service (requirements, page 2).

Get Inventory

Using the Visitor Pattern (*Head First Design*, appendix), traverse all Authentication Service objects to compile an inventory of all Resources, Users, Roles, Permissions, and AuthTokens (assignment 4 handout, page 2).

Login

A User can login using a Credential (password, face print, voice print). Admins have a password on the account (used when running a script through the CommandProcessor). All Users have a face print, voice print, or both that are assigned when the User is created (requirements, page 2). A non-Admin User logs-in when they enter the Store, which identifies and authenticates them.

Identify and Authenticate User

When entering a Store (or logging in with a password in the case of the Admin running a script), the Authentication Service looks up the User and compares the Credential with that stored in the system. If the Credentials match, an AuthToken is generated and associated with the User.

Create AuthToken

If a User logs-in, is identified and authenticated, the Authentication Service generates a new AuthToken that is associated with the User. The Token includes an expiration time and an active status

Logout

A User logs out when leaving the Store (or manually if the User is an Admin running a script). On logout, the Authentication Service invalidates the AuthToken associated with the User.

Invalidate AuthToken

If a User logs-out, the AuthToken associated with their object is invalidated. The Authentication Service updates the “active” status to False to prevent further use. An AuthToken can also be invalidated if it has not been used within a given period of time — i.e. timeout (requirements, page 5).

Invoke Restricted Method

The StoreControllerService, Admin (through a script run in the CommandProcessor), or other User’s can call on the StoreModelService public API by passing in their AuthToken. All Model Service calls include a “authToken” parameter which start by checking the Authentication Service to see if the permissions allow the method to be called.

Check Access

When invoking a restricted method, the Store Model Service passes the AuthToken it receives to the Authentication Service to check access permissions. It uses the AuthVisitor to check all Entitlements to verify the User has the appropriate permissions to make the restricted method call.

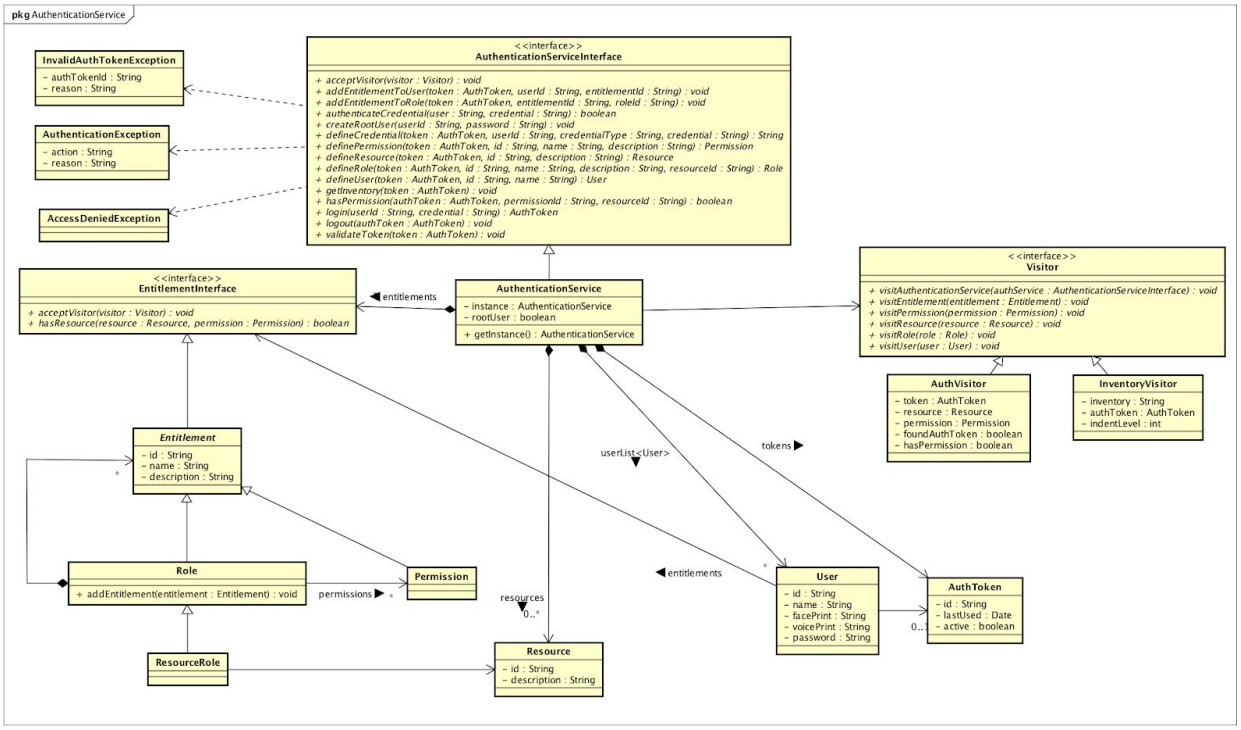
Implementation

Class Diagram

The following class diagram defines the Authentication Service implementation classes contained within the package “com.cscie97.store.authentication”.

The CommandProcessor, while it interacts with the Authentication Service, is part of the Controller Service. This interaction can be seen in the component diagram in the “Overview” section above, and further information can be found in the “Implementation Details” section below.

Note: full-resolution images can be found in the zipped submission folder under the “Diagrams” directory.



Class Dictionary

This section specifies the class dictionary for the Authentication Service. The classes are defined within the package “com.cscie97.store.authentication”.

AuthenticationServiceInterface

The main interface that is implemented by a concrete Authentication Service. The interface defines the basic functionality for the service, including creating entities (User, Entitlements, Resources) and assigning Entitlements to Users.

A concrete instance of the AuthenticationServiceInterface also accepts Visitors, following the Visitor pattern as described in *Head First Design*, Chapter 9. See the “Visitors” section below for more details.

Methods

Method Name	Signature	Description
acceptVisitor	(visitor : Visitor) : void	Accept a Visitor. Actions and navigation follow the Visitor instance definition.
addEntitlementToUser	(AuthToken token, userId : String, entitlement : Entitlement) : void	Associate an Entitlement to the specified User.
addEntitlementT	(AuthToken token,	Associates an Entitlement with the specified

oRole	entitlement:Entitlement, role:Role) : void	Role.
authenticateCredential	(userId: String, credential : String) : boolean	Compare the provided credential with the one stored for the specified User. The credential is checked against all three potential stored values: voice print, face print, and password. To check a password credential, the provided String is hashed and compared with the stored hashed password (requirements, page 6). An valid authentication will return true, otherwise false.
createRootUser	(userId: String, credential:String):void	To address the bootstrapping problem, as outlined in Piazza post @267, the Authentication Service allows for a single root user to be initialized with admin Permission. Once created, an admin Permission must be passed to all subsequent API calls to the Authentication Service.
defineCredential	(AuthToken token, userId: String, credentialType : String, credential:String) : void	Create a new credential for the specified User and associate with the User. The credentialType can match one of three options: “voice_print”, “face_print”, or “password”. A User can have only one kind of credential at a time. If a new one is provided, the old credential is overwritten. If the credentialType matches “password”, the credential value is hashed, and that hash is stored with the User instead of the plain text password.
definePermission	(AuthToken token, id : String, name: String, description: String) : Permission	Create a new Permission object with id, name, and description.
defineResource	(AuthToken token, id : String, description: String) : Resource	Create a new Resource object with id and description.
defineRole	(AuthToken token, id : String, name: String, description: String,	Create a new Role with id, name, and description. If the resourceId is not null, the specified Resource is retrieved and

	resourceId : String) : Role	associated with the Role by creating a ResourceRole.
defineUser	(AuthToken token, id : String, name: String) : User	Create a new User with id and name.
getInventory	(token:AuthToken):void	Construct an inventory of all Authentication Service objects. This creates an InventoryVisitor which will traverse the structure to compile and print an inventory of all defined objects.
hasPermission	(token:AuthToken, permissionId:String, resource:Resource):boolean	Check the AuthToken for a specified Permission. An AuthVisitor is created to look for the User associated with the AuthToken. If a User is found, their Entitlements are checked for the specified Permission.
login	(userId:String, credential:String):AuthToken	Login the User with provided credentials. The credential is authenticated with those stored with the User. If valid, a new AuthToken is created to verify restricted API calls.
logout	(authToken:AuthToken):void	Invalidate the AuthToken provided.
validateToken	(token:AuthToken):void	Validate the provided AuthToken. Checks whether a token is provided, whether that token is active, and if it was issued within the TOKEN_TIMEOUT duration. As noted in the comments, TOKEN_TIMEOUT is specified in a duration of milliseconds to support testing for assignment 4. For a production application, a longer duration (such as minutes or hours) would probably be more appropriate.

AuthenticationService

A concrete AuthenticationSerivce that implements the AuthenticationServiceInterface to use with the CommandProcessor (described in the Assignment 3 design document; additional details described below in the “Implementation Details” and “Testing” sections).

This concrete Authentication Service defines an additional method, getInstance(), to adhere to the Singleton Pattern described in *Head First Design*, Chapter 5 (assignment 4 handout, page 2). The Authentication Service contains a private constructor which is called once, only by this getInstance() method. Subsequent calls return the same instance of the class. This is important

for the Authentication Service to prevent varied states depending on when a public API call is made.

The AuthenticationService is responsible for defining all Authentication entities (Users, Entitlements), assigning Permissions to Users, and generating AuthTokens when validating a User's Credentials.

Properties

Property Name	Type	Description
instance	AuthenticationService	The instance of the class.
rootUser	boolean	Whether a root user has been initialized or not. Once the createRootUser() method is run, this property is set to true and cannot be run again.

Methods

Method Name	Signature	Description
getInstance	() : AuthenticationService	Instantiates a new object the first time it is called. Afterwards, when getInstance() is called, it refers to the previously instantiated instance.

Associations

Association Name	Type	Description
userMap	Map<String, User>	A map of Users that exist within the AuthenticationService.
permissionMap	Map<String, Permission>	A list of all Permissions created by the service.
roleMap	Map<String, Role>	A list of all Roles created by the service.
resourceMap	Map<String, Resource>	A list of all Resources created by the service.
tokenMap	Map<Integer, AuthToken>	A list of all AuthTokens created by the service.

Visitor

This interface follows the Visitor pattern as described in the appendix of *Head First Design*. The Visitor is constructed with knowledge of the Authentication Service and contains methods to visit each of the various Authentication Service entities. Traversal is performed by logic defined in concrete Visitor classes. Traversal begins in each Visitor when passed through the acceptVisitor() method in the Authentication Service.

Methods

Method Name	Signature	Description
visitAuthenticationService	(authenticationService:AuthenticationServiceInterface):void	Visits an Authentication Service interface. Will perform action(s) needed and then traverse down the tree to additional objects.
visitEntitlement	(entitlement:Entitlement):void	Visits an Entitlement. Will perform action(s) needed and then traverse down the tree to additional Roles or Permissions.
visitPermission	(permission:Permission):void	Visits a Permission. Will perform action(s) needed and then return control back to the parent Role.
visitResource	(resource:Resource):void	Visits a Resource. Will perform action(s) needed and then return control back to the Visitor.
visitRole	(role:Role):void	Visits a Role. Will perform action(s) needed and then return control back to the Visitor.
visitUser	(user:User):void	Visits a User. Will perform action(s) needed on the User and/or associated AuthToken/Credentials.

AuthVisitor

A concrete Visitor class that implements the Visitor interface. The AuthVisitor traverses a User's list of Entitlements to determine whether or not the User has the specified Permission (assignment 4 handout, page 2). Traversal logic is handled by the Visitor, as defined via the interface methods above.

Properties

Property Name	Type	Description
token	AuthToken	The token to check Permissions of. It is

		associated with a User who has a list of Entitlements.
resource	Resource	Optionally, a Resource that can restrict the search for valid Permissions.
permission	Permission	The Permission to search for.
foundAuthToken	boolean	Indicator for whether the AuthToken is found.
hasPermission	boolean	Indicator for whether the AuthToken contains access to the Permission.

Methods

Method Name	Signature	Description
hasPermission	():boolean	True if the User's list of Entitlements contains the requested Permission. False otherwise.

InventoryVisitor

A concrete Visitor class that implements the Visitor interface. The InventoryVisitor traverses all AuthenticationService objects to construct and log a full inventory. No additional methods are required as the traversal and printing logic is implemented in the visitEntity() methods defined in the Visitor interface.

Properties

Property Name	Type	Description
inventory	String	The Inventory that is built by the Visitor as it traverses the Authentication Service structure.
authToken	AuthToken	Token to make restricted API calls.
indentLevel	Integer	The current indent level, used for formatting the output of the Inventory.

EntitlementInterface

The main interface that is used to implement the Composite Pattern (as described in *Head First Design*, Chapter 5). Entitlements are created by the Authentication Service, with a Role being a composite of Permissions (requirements, page 5). Permissions are the leaf nodes of the system.

ResourceRoles are a specific kind of Role that is associated with a Resource.

No methods are defined by the Entitlement interface, but to adhere to the Dependency Inversion Principle, an interface is defined to allow for easier expandability in the future and to maintain best practices. Properties that are similar among all Entitlements are defined in the abstract *Entitlement* class.

Entitlement

An abstract Entitlement class that implements the Entitlement Interface. defines properties common for all Entitlement classes. Each Entitlement contains an id, name, and description.

Properties

Property Name	Type	Description
id	String	The unique id of the Entitlement.
name	String	The name of the Entitlement.
description	String	A description of the Entitlement.

Role

The Role class extends the Entitlement class. A Role is a specific kind of Entitlement that is a composite of Permissions. Roles group Permissions and Roles together to more easily grant multiple permissions.

Methods

Method Name	Signature	Description
addEntitlement	(entitlement:Entitlement):void	Add the specified Entitlement to the Role. The method checks the existing list to prevent duplicate Permissions from being added.

Associations

Association Name	Type	Description
entitlements	List<Entitlement>	A list of Entitlements associated with the Role.

Permission

The Permission class extends the Entitlement class. A Permission represents a logical permission that grants access to a specific resource or function of the Store system

(requirements, page 5). It is fully defined by the Entitlement class with the constructor passing all parameters through to the super() method.

ResourceRole

The ResourceRole extends the Role class, further extending the Entitlement class. A ResourceRole is a specific form of Role, that ties the Entitlements composed by the ResourceRole to the specific Resource associated with the class.

Associations

Association Name	Type	Description
resource	Resource	The Resource that the Role is tied to.

Resource

A Resource is a representation of a physical entity within the Store 24X7 System. Examples include a Store, Sensor, or Appliance.

Properties

Property Name	Type	Description
id	String	The unique ID of the resource.
description	String	A description of the Resource.

User

A User is the AuthenticationService representation of a person. In the case of the Store 24X7 System, this corresponds to a Customer in the Store Model Service and an Account in the Ledger Service. A User is associated with credentials to uniquely identify them and verify the User. One of each credential type is allowed.

Properties

Property Name	Type	Description
id	String	The User's unique ID.
name	String	The name of the User.
login	String	The User's login credential, which stores a hashed password.
facePrint	String	The User's face print, simulated using the format "--face:<username>--".

voicePrint	String	The User's voice print, simulated using the format "--voice:<username>--".

AuthToken

An authorization token grants the User the permissions associated with their account. An AuthToken is generated after a successful login/face/voice verification.

Properties

Property Name	Type	Description
id	String	The unique id of the AuthToken.
lastUsed	Date	The time the AuthToken was last used. If the token hasn't been used for "TOKEN_TIMEOUT" duration, the token is marked as invalid and the User must request a new token.
active	boolean	A True/False indicator of whether the AuthToken is active or not. True, if the AuthToken is valid, False otherwise.

Event Syntax

Add Entitlement to Role

```
add permission_to_role <permission_id> <role_id>
```

Add Entitlement to User

```
add entitlement_to_user <user_id> <entitlement_id>
```

Create Root User

```
create auth_root_user <user_id> <password>
```

Define Credential

```
define credential <user_id> (voice_print | face_print | password) <value>
```

Define Permission

```
define permission <permission_id> <permission_name> <permission_description>
```

Define Resource

A new Resource is created when the "define store" command is run in the Store Model Service set up. The Store Model syntax is copied here. Only the id and name are used (the name is used in place of the resource description.

```
define resource <resource_id> <resource_description>
```

Define Role

```
define role <role_id> <role_name> <role_description> [<resource_id>]
```

Define User

A new User is created when the “define customer” command is run in the Store Model Service set up. The Store Model syntax is copied here. Only the id, first and last names are used for a Authentication Service User.

```
define user <user_id> <user_name>
```

Get Inventory

```
get auth inventory
```

Login

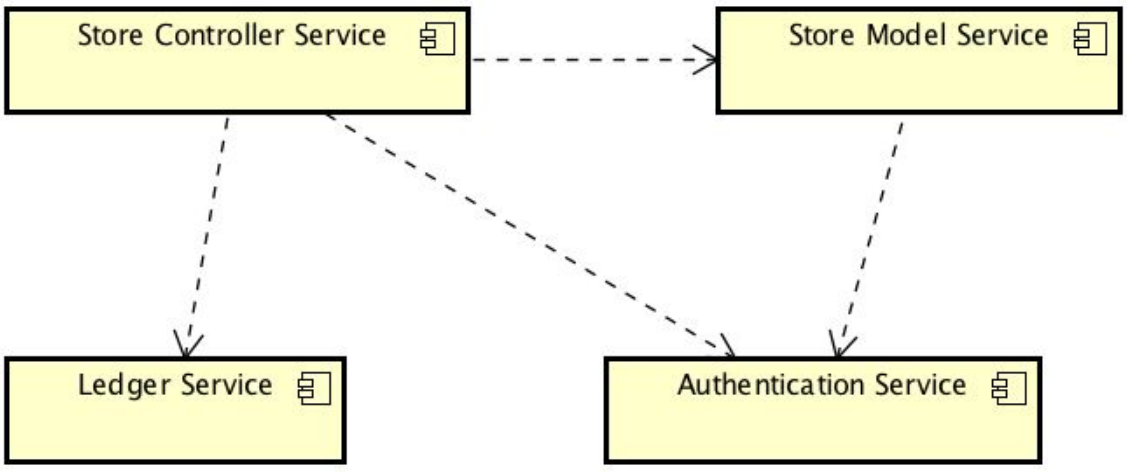
```
login user <user_id> password <credential>
```

Logout

```
logout user <user_id>
```

Implementation Details

The Authentication Service fits into the larger Store 24X7 System and integrates with two other components — the Store Controller Service and Store Model Service. The following diagram from the assignment 4 handout illustrates how these services connect:

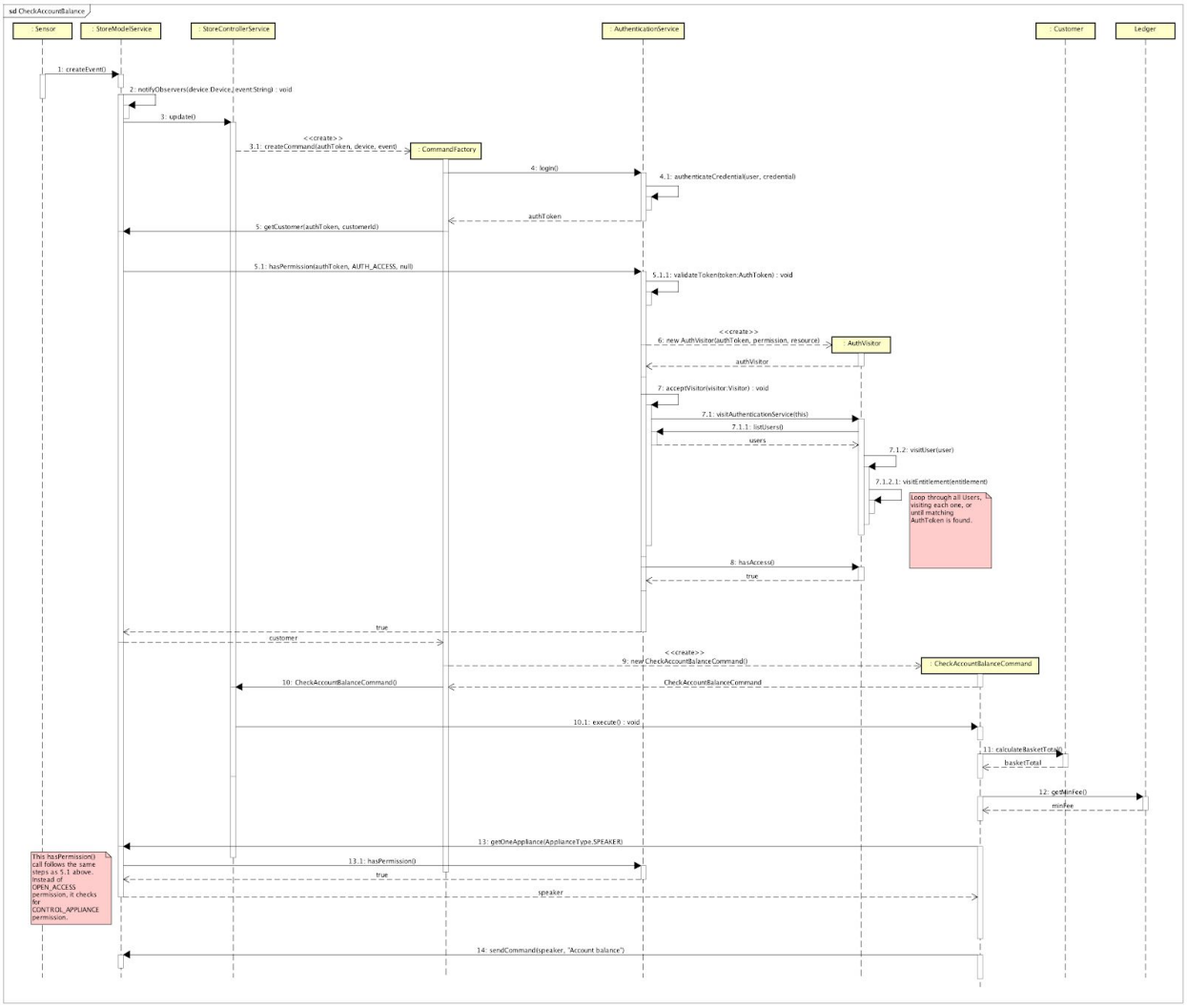


For more information about the Ledge Service, Store Model Service, and Store Controller

Service, reference the design documents for assignments 1-3 which are included in the submission folder.

The Authentication Service follows three design patterns from the *Head First Design* book: the Visitor Pattern (appendix), Singleton Pattern (chapter 5), and the Composite Pattern (chapter 9).

The Authentication Service uses the Visitor Pattern to support two use cases: 1) construct an inventory of Authentication objects; and 2) check for access. The Visitor interface outlines several visit() methods, one for each type of object defined in the Authentication Service — the Authentication Service itself and methods for Entitlement, Permission, Resource, Role, and User objects. To support the two use cases, two concrete Visitor classes — the AuthVisitor and InventoryVisitor — are defined to handle the traversal needed.



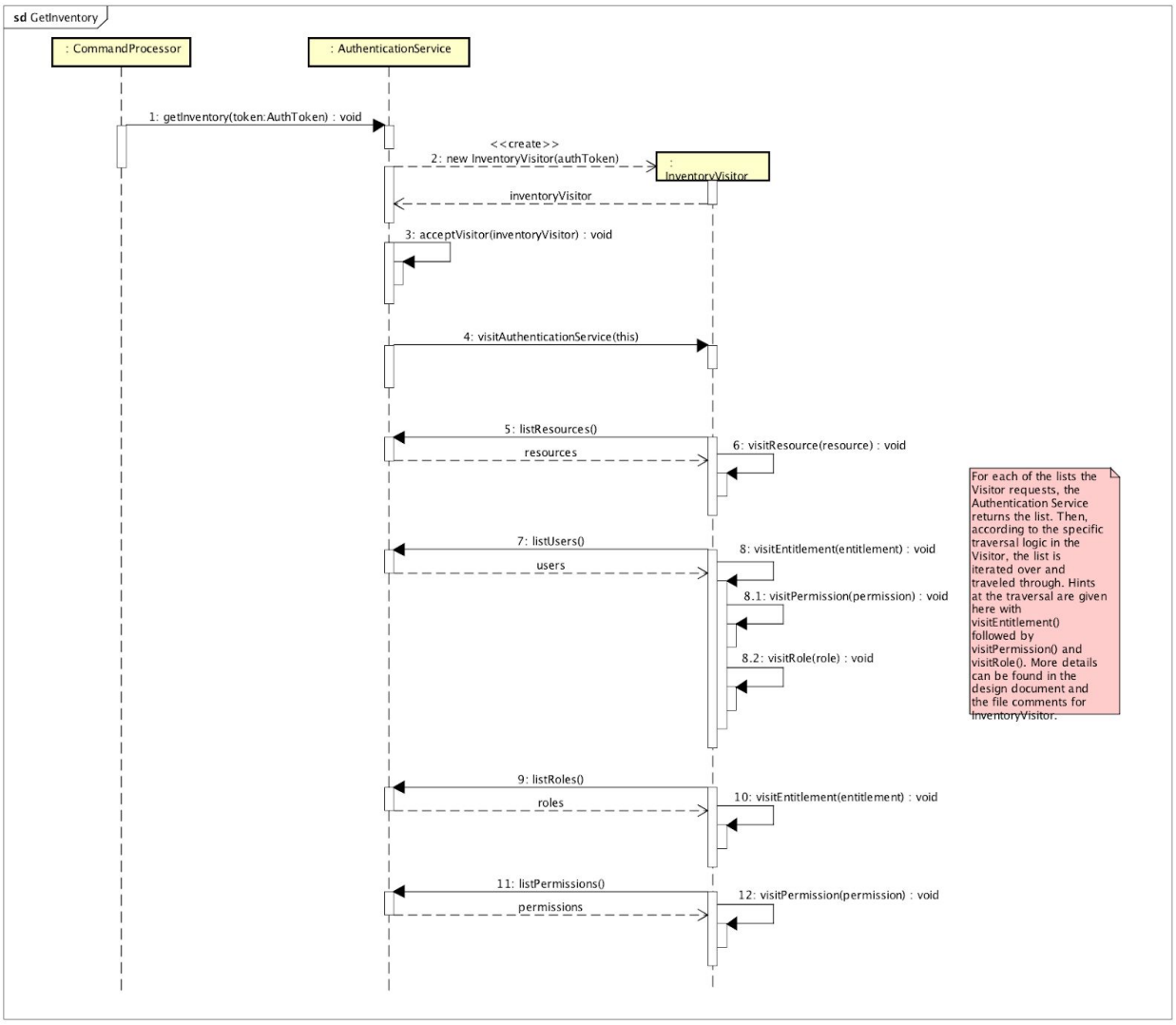
This sequence diagram extends the CheckAccountBalance sequence diagram from assignment 3. This sequence diagram integrates the creation of an AuthVisitor to check the AuthenticationService for access. (Note: full-resolution images can be found in the zipped submission folder under the “Diagrams” directory).

Functionality that was added for the Authentication Service includes steps 4-8 and 13.1 in the sequence diagram. Starting at step 4, the CommandFactory must first log in the user referenced in the event. The AuthenticationService authenticates the provided credential and returns an AuthToken back to the CommandFactory. This token is then passed to the getCustomer() call to the Store Model Service. In assignment 3, the Store Model Service returned the Customer directly. In assignment 4, it first calls the Authentication Service to check hasPermission() for the “AUTH_ACCESS” permission.

The Authentication Service then goes through several steps to validate the token, create a new AuthVisitor, and then accept the Visitor to check for access. To check for access, the Visitor iterates over the list of Users in the Authentication Service until it finds the User with the matching AuthToken. If found, the Visitor iterates over the User’s Entitlements checking for the “AUTH_ACCESS” permission.

After the traversal is complete, the Authentication Service checks the result by calling hasAccess() on the Visitor, which returns a boolean value. If true, the Store Model Service then grants access and returns the Customer back to the Command Factory. With the information it needs, the Factory then creates the CheckAccountBalanceCommand as in assignment 3.

When the Store Controller Service calls execute() on the command, the Command performs the specified steps. The only restricted task is the receiveCommand() method sent to the speaker Appliance. In order to validate permissions, the Store Model Service calls hasPermission() on the Authentication Service again, passing the AuthToken to see if contains the “CONTROL_APPLIANCE” permission. If true, the speaker can tell the Customer their account balance.



This sequence diagram walks through the InventoryVisitor constructing the inventory of the Authentication Service objects.

In order to implement the Singleton Pattern, the constructor for the AuthenticationService is defined as private such that it can only be called by the Authentication Service itself. In order for an outside class to access the Service, a getInstance() method is provided. This method, when called for the first time, instantiates a new Authentication Service by calling the private constructor. On subsequent calls, the same instance is returned to the caller, thereby ensuring that only one instance of the Service can exist.

Several classes exist to follow the Composite Pattern as outlined in *Head First Design* and the assignment 4 requirements (page 5). An Entitlement class (which implements the EntitlementInterface) defines properties common to all of the composite objects. A Role is a composite class that can contain other Entitlements (Roles and Permissions). A ResourceRole

extends the main Role and provides a further restriction by tying the Entitlements to a specific Resource. Finally, Permissions are the “leaf nodes” in the pattern and represent the individual permissions needed to call restricted methods.

Exception Handling

AuthenticationException

The AuthenticationException is returned from the Authentication Service in response to an error condition. The AuthenticationException captures the action being performed and the reason for failure.

Properties

Property Name	Type	Description
action	String	The action that was performed (e.g. define permission)
reason	String	The reason for the exception (e.g. A Permission with id ‘xx’ already exists.)

AccessDeniedException

The AccessDeniedException is returned from the AuthenticationService when the User requesting access does not have the required Permissions to make the public API call to the Store Model Service.

Properties

Property Name	Type	Description
action	String	The action that was called (e.g. “add basket item”)
permission	String	The Permission needed to make a public APi call to the Store Model Service. (e.g. “basket_items”)

InvalidAuthTokenException

The InvalidAuthTokenException is returned from the Authentication Service in response to a failed check when trying to validate the token. This Exception is returned whenever the “active” property of the AuthToken is set to False, and can be due to timeout, or the User logging out of the system, thereby invalidating the token.

Properties

Property Name	Type	Description
authTokenId	String	The invalid AuthToken.
reason	String	The reason for the Exception. (e.g. the token timeout duration has been exceeded and the token is now invalid).

Testing

A TestDriver class will be defined within the package “com.cscie97.store.test” and will instantiate a Ledger Service, Store Model Service, Store Controller Service, and Authentication Service. The TestDriver supports reading in one or more test script(s) (authentication.script and others are provided), and handing off the instructions to the CommandProcessor. The CommandProcessor accepts input relating to all Service public API methods to load a pre-defined store configuration, account information, and authentication information.

To address the Authentication bootstrapping problem, as mentioned in Piazza post @267, the Authentication Service supports a “create auth_root_user” command to bypass permission restrictions on creating Authentication objects. Only one root user can be created, and once created, valid “admin” permissions are needed to create additional Authentication objects and validate calls to Store Model Service public API methods.

To simplify some of the overlap between the Authentication Service and Store Model Service, as hinted at in Piazze post @296, the Store Model Service commands “define customer” and “define store” have been modified in the CommandProcessor to also create User and Resource objects respectively in the Authentication Service. Separate “define user” and “define resource” commands do not exist in the script syntax for the CommandProcessor.

Functional testing is performed by validating that executed commands output correct results (e.g., If a “define permission” or “add role to user” commands are issued, the Authentication Service creates the objects. Negative testing is also performed to validate that the Authentication Service ignores unrecognized commands as well as if invalid data is passed through the command (e.g., If a ‘define user’ event is issued but the requested ID is not unique, an error message should be logged).

Performance testing is not handled for this assignment as we do not have all the tools to handle doing so. As noted in the design document for assignments 2 and 3, performance testing could be achieved by, “writing a long test script or adjusting the CommandProcessor to accept multiple script files in succession to achieve a kind of scale we might anticipate in production.” However, for the purposes of this assignment, provisioning a Store and a small number of related store entities, Customers, Ledger Accounts, and Authentication Users and Entitlements will be sufficient to test the requirements outlined and ensure the Authentication Service works

as intended.

For regression testing, since parts of the Store Controller and Model Services are updated to incorporate Authentication Service, the same test script that was submitted in assignments 2 and 3 should be run again with the output compared to the previous implementation. Any differences or variations that are detected should be addressed and modified to produce identical output.

Exception handling is described in its own section above.

Risks

Implementing the Authentication Service addresses a big risk that was present in the Store 24X7 System by including AuthToken's. While the design as written takes many principles of security into consideration (such as hashing passwords instead of storing in plain text), hackers are resilient and in a system dealing with physical goods and monetary transactions, the utmost care should be practiced.

Man-in-the-middle and denial-of-service attacks are two risks inherent to the system. Considering persistence is not a requirement for this assignment, a hacker could run a denial-of-service to crash the system, gain control, and then re-provision the Store 24X7 System to their desire. A man-in-the-middle attack could be run (perhaps in conjunction with a DoS attack) to intercept AuthTokens and User credentials to impersonate Users. A physical hack could also be employed, which, depending on the sophistication of our physical Cameras and Microphones, could be exploited using taped voice recordings or 3D printed masks (Mission: Impossible style).