Store Model Service Design Document

Date: 10/9/19
Author: Gerald Arocena
Reviewer(s): Nicholas Antwi, Sean Bollin

After the Introduction and Overview below, this design document contains the following sections (in order of appearance): Requirements (pg. 2), Use Cases (pg. 4), Implementation (pg. 7), Implementation Details (pg. 19), Exception Handling (pg. 19), Testing (pg. 20), and Risks (pg. 20). The Requirements section recapitulates the Store Model Service requirements. The Use Cases section includes a use case diagram and explains how different actors interact with the system. The Implementation section includes a class diagram that is followed by a class dictionary.

Introduction

IoT is set to be ubiquitous in daily life. The grocery store industry is one area that IoT is planned to impact in the coming years. As IoT technology such as RFID has been making a comeback in the retail industry in general over the past couple years, grocery stores are only set to become more IoT-driven.

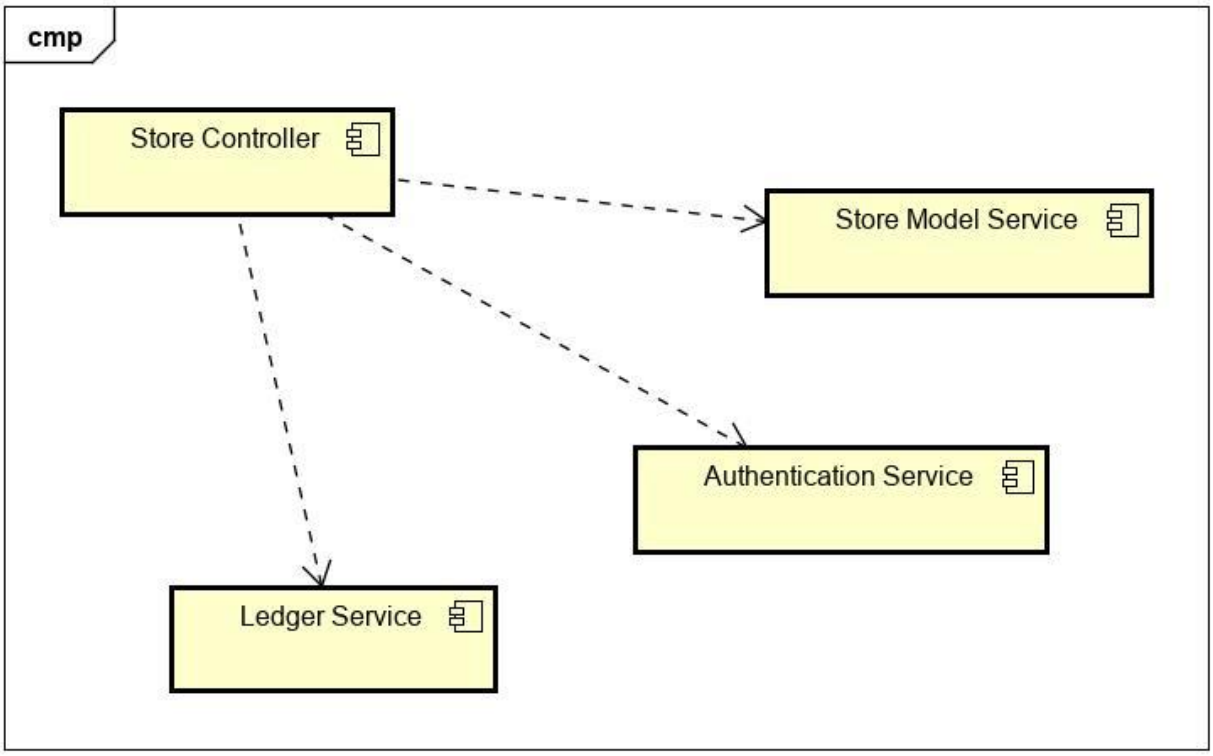
This design specification explains how to implement a Store Model Service, a cloud-based service. The Model Service provisions and maintains the assets and information for a store including customers and inventory, and for multiple stores. Other assets include aisles, shelves, and IoT sensors and appliances. Sensors and appliances collect and share data such as customer location and appliances can be controlled. The Store Model Service is able to send events to both sensors and appliances, and commands to appliances.

Overview

In the zeitgeist of the trending Internet of Things / “smart” things paradigm, the Store 24X7 System seeks to make the “store of the future” fully automated. In doing so, it employs modular design based on the three design principles of divide and conquer, abstraction, and separation of concerns and thus making the implementation process more manageable. The finalized system would allow for a store to be fully automated by the 24X7 software. For instance, customers could simply walk past sensors as they exit the store with items they wish to purchase and the 24X7 System would handle any transaction processing needed autonomously.

The Store Model Service represents one of the four modules that make up the Store 24X7 System. It implements an API interface that allows for provisioning, maintaining, and updating stores and their assets. Any external party (e.g., the Store Controller, a testing script) that needs to communicate with the Store Model Service must do so through the API interface. The diagram on the following page illustrates the relationship between the components that make up

the system.



Caption: The four major modules that comprise the Store 24X7 System architecture are shown above. The Ledger Service is used for customer transactions. The Store Model Service manages the state of the store and its assets. The Authentication Service manages access to the store and its assets including the control of appliances. The Store Controller must listen for store events and respond appropriately, and check out customers. Thus, it needs to interact with all of the other three modules.

Requirements

The following lists briefly summarize major parts of the Store Model Service requirements:

Provisioning Stores

1. The Model Service must be able to process provisioning commands for configuring a store's layout, architecture, inventory, customers, and other assets.
2. The service must be able to access, collect, share, and manipulate the state of assets in a store.
3. It must be able to manage multiple stores.

Stores

1. The Model Service will manage a list of stores.
2. A store must be assigned a globally unique id.
3. Additional store assets include inventory, products, customers, sensors, and appliances.

Store Assets

1. Product id is assigned by an external source, i.e., not by the store.
2. Inventory defines the relationship between the products a store sells and their locations on shelves. Product count must stay between zero and capacity inclusive.
3. Registered customers must have identity information provided to the stores, can remove items from the stores (guests cannot), and are recognized by all stores.
4. Sensors and appliances must be able to accept events.
5. Appliances must be able to accept commands.

Store Model Service

The Store Model Service must be able to:

1. Define a store including its assets.
2. Get a store's details.
3. Create/simulate sensor and appliance events.
4. Send commands to appliances.
5. Access the state of sensors and appliances (and other assets), and update/control state if needed.
6. Monitor and support customers.
7. The service needs to be compatible with a Command Line Interface (CLI) that has its own domain-specific language (DSL). Command syntax for this language can be seen in the class dictionary under the CommandProcessor class.
8. All API methods must include a "auth_token" parameter to support access control.

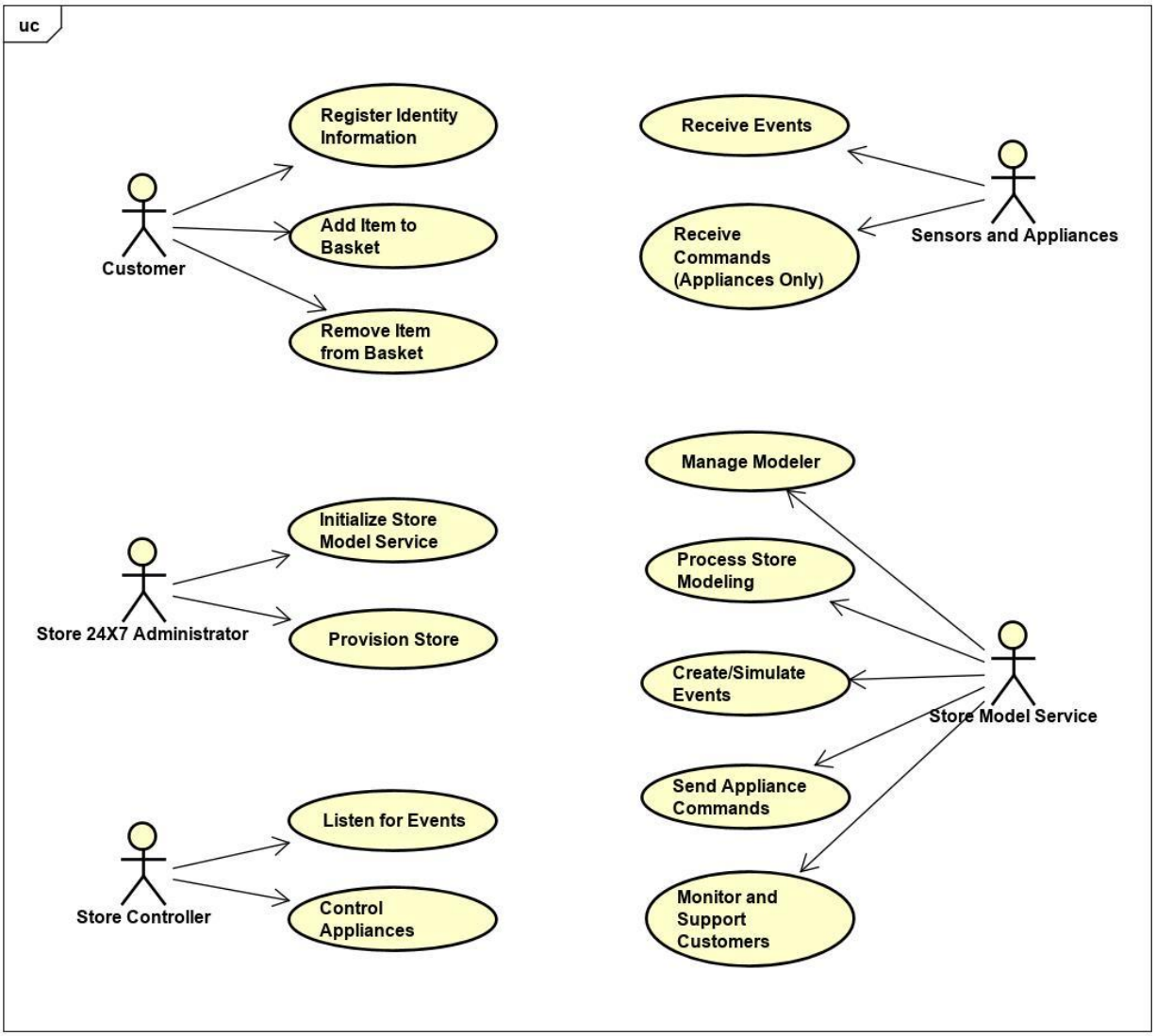
Not Required

Persistence

The Store Model Service does not require anything to be put into long-term memory to operate. It is fully functional with RAM or similar.

Use Cases

The diagram below illustrates the use cases of the Store Model Service.



Caption: Shows the five actors in the Store Model Service. Each of their use cases are shown describing how each actor interacts with the system.

Actors:

The actors of the Store Model Service System are the Customer, Store 24X7 Administrator, Store Controller, the Sensors and Appliances, and the Store Model Service.

Customer

Customers are able to shop with a basket at any of the stores serviced by the Model Service. They can also register their personal, billing, and other information with the store service.

Store 24X7 Administrator

The 24X7 Administrator is able to initialize the Modeler – a singleton class that implements the store Service Model's API interface – and create, provision, and manage stores.

Store Controller

The store controller listens for events and responds accordingly. For instance, when a camera senses a customer putting an item in their basket, it generates a sensor event that the store controller responds to by accounting for how much of the item was removed and which shelf it was taken from. The Store Controller will be implemented in the next module. It will use the Store Model Service's API interface to manage and operate stores through their sensors and appliances.

Sensors and Appliances

Sensors and appliances are able to receive events. Appliances can also receive commands.

Model Service

The Model Service is autonomous and is able to manage the Modeler, the provisioning of stores, and manage store activity. It can create and send events to the sensors and appliances and create and send commands to appliances.

Use Cases:

Initialize Store Model Service

The initialization of the Modeler system is built-in indirectly. That is, if an API command is sent to it, the Modeler class will create a Modeler object automatically if one doesn't already exist.

Provision Store

The Store 24X7 Administrator must be able to create or acquire a store and any assets associated with it including different types of sensors and appliances, products for selling, and different types of shelf containers as well as manipulate the assets in a store to some extent including the amount of each asset, where they're placed, how they're identified, temperature settings and similar such things. They should have ready access to the details of a store and its assets including products that are being sold and their associated information, inventory information and other such things.

Register Identity Information

Customers must be able to register identity, billing, and other information about themselves to the Store Model Service.

Add Item to Basket

Registered customers are be able to add store products to the virtual basket that's assigned to them upon entering the store.

Remove Item from Basket
Registered customers are be able to remove items from the virtual basket that's assigned to them upon entering the store.

Listen for Events
In future modules, the Store Controller will listen for sensor and appliances events in order to manage store activities.

Control Appliances
In future modules, the Store Controller will be able to react to store events by sending commands to appliances.

Receive Commands
Appliances must be able to receive commands.

Receive Events
Sensors and appliances must be able to receive events. Events for the Store Model Service are simulated and not actually an account of events happening in a physical store that exists in reality. As per the requirements, appliances must be able to receive events in order to test the implementation of the store 24X7 system.

Manage Modeler
The Model Service makes sure that store modeling is handled correctly and for multiple stores. Each store functions and operates with its own independent identity and must be protected from undue interference from other stores' operations or any other outside interference. Each store must be able to access products appropriately, pick and choose which products to sell, have autonomy in creating inventory and in allocating sensors and devices in their aisles, among other things. In addition, customers must be able to shop at any store.

Process Store Modeling
As the Store Model Service processes API method calls, it should make sure that all requirements are met throughout. For example, it must ensure that Inventory and other domain objects' identifiers are unique (respective of scope) before creating them, and that only registered customers are assigned baskets. The Model Service makes sure that assets are accessible (or lack thereof) and that they're updated appropriately. For instance, it ensures that inventory product counts stay within their range limits, and that guests can't remove items from the store (only registered customers can) as per the requirements.

Create/Simulate Events
The Model Service is be able to create events and send them to specific sensors and appliances. As far as this Store Model Service module implementation is concerned, events can be thought of as opaque strings. Events will be redefined in later modules.

Send Appliance Commands

The Model Service is be able to create and send commands to specific appliances. As far as this Store Model Service module implementation is concerned, commands can be thought of as opaque strings. Events will be redefined in later modules.

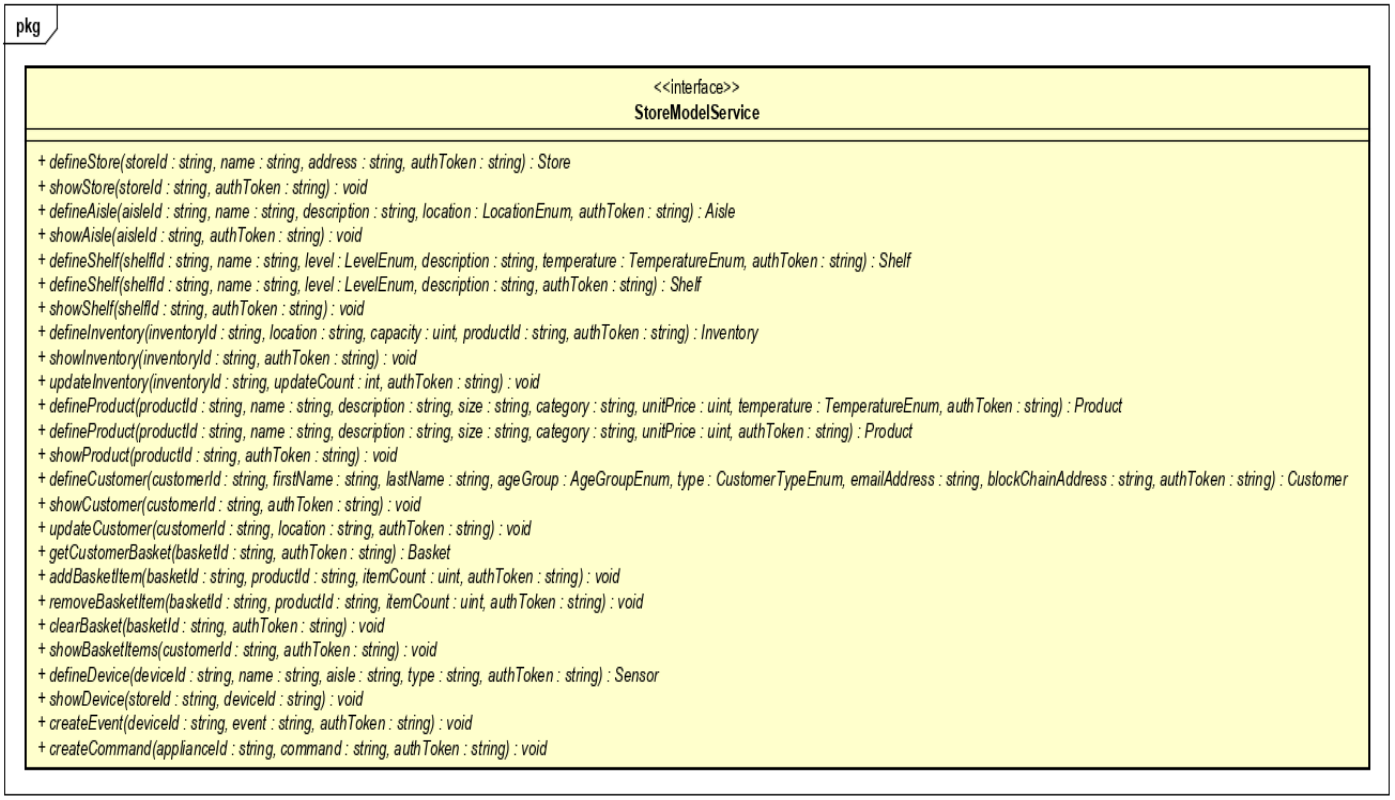
Monitor and Support Customers

The model service is be able to access and update a customer's location and activity at any store, e.g., the number of items in their basket. It must retrieve identity and other information on registered customers so that they can be identified, and billed. In future modules, customers, sensors, and appliances (e.g., human-like robots, and speakers) will be much more interactive with one another.

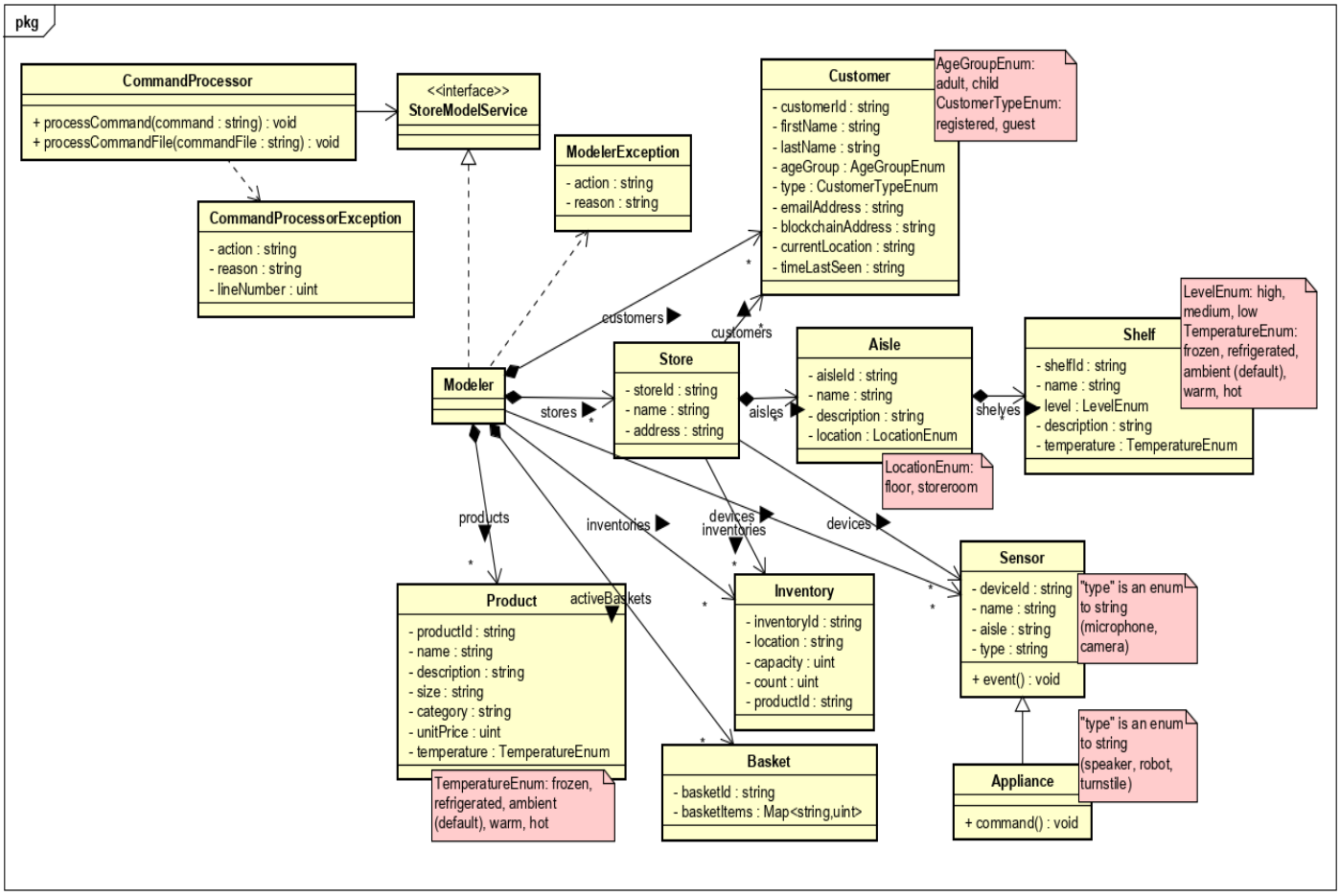
Implementation

Class Diagram

The following are the Store Model Service interface and class diagram that define the implementation classes and comprise the package "com.cscie97.store.model".



Caption: The API for the Store Model Service interface. In the accompanying class diagram on the next page, the methods shown above are excluded in the interest of spacing out elements on the page for readability.



Caption: Class diagram for the Store Model Service. The notes list the enumeration values the class uses.

Class Dictionary

This section contains the class dictionary for the Model Service that comprises the package “com.cscie97.store.model”.

Modeler

The Modeler provides the API of the Store Model Service for clients to use. It handles the provisioning and management of stores and their assets including their state and information. The requirements state that the Model Service must provide a public API for handling such tasks and this class serves that purpose.

Associations

Association Name	**Type**	**Description**
stores	map<string, Store>	The mapping of store id’s to stores under

		the Modeler's service. The requirements state that it must be able to manage multiple stores.
customers	map<string, Customer>	The mapping of customer id's to customers that can shop at any store.
products	map<string, Product>	The mapping of product id's to products that any store can sell.
inventories	map<string, Inventory>	The mapping of inventory id's to inventories from all stores.
activeBaskets	map<string, Basket>	The mapping of basket id's to assigned customer baskets.
devices	map<string, Sensor>	The mapping of device id's to sensors from all stores.

Methods

Method Name	**Signature**	**Description**
defineStore	(storeId : string, name : string, address : string, aisle : Aisle, shelf : Shelf, authToken : string) : Store	Creates a new Store. Store must have a globally unique id. Returns the Store object.
showStore	(storeId : string, authToken : string) : void	Shows the details of a store with given id in stdout.
defineAisle	(aisleId : string, name : string, description : string, location : enum, authToken : string) : Aisle	Creates a new Aisle. Returns the Aisle object.
showAisle	(aisleId : string, authToken : string) : void	Shows the details of the aisle with given id in stdout.
defineShelf	(shelfId : string, name : string, level : enum, description : string, temperature : enum, authToken : string) : Shelf	Creates a new Shelf. Returns the Shelf object.
defineShelf	(shelfId : string, name : string, level : enum, description : string, authToken : string) : Shelf	Creates a new Shelf with a default temperature ("ambient"). Returns the Shelf object.
showShelf	(shelfId : string, authToken : string) : void	Shows details of the shelf with given id in stdout.
defineInventory	(inventoryId : string, location : string,	Defines a new Inventory.

	capacity : uint, productId : string, authToken : string) : Inventory	Returns the Inventory object.
showInventory	(inventoryId : string, authToken : string) : void	Shows inventory information for given inventory id in stdout.
updateInventory	(inventoryId : string, updateCount : int , authToken : string) : void	Updates Inventory with the given id and count. Uses a negative number if decrementing count.
defineProduct	(productId : string, name : string, description : string, size : string, category : enum, unitPrice : uint, temperature : enum, authToken : string) : Product	Defines a new Product. Returns Product object.
defineProduct	(productId : string, name : string, description : string, size : string, category : enum, unitPrice : uint, authToken : string) : Product	Defines a new Product with default temperature (“ambient”). Returns Product object.
showProduct	(productId : string, authToken : string) : void	Shows the details of product with given id in stdout.
defineCustomer	(customerId : string, firstName : string, lastName : string, type : enum, emailAddress : string, blockChainAddress : string, authToken : string) : Customer	Creates a new Customer. Returns Customer object.
showCustomer	(customerId : string, authToken : string) : void	Shows details of customer with given id in stdout.
updateCustomer	(customerId : string, location : string, authToken : string) : void	Updates location information of Customer with given id.
getCustomerBasket	(basketId : string, authToken : string) : Basket	Retrieves the basket for a customer. If customer doesn’t already have one, one is created.
showBasketItems	(customerId : string, authToken : string) : void	Shows the items in a customer’s basket with given id in stdout.
addBasketItem	(basketId : string, productId : string, itemCount : uint, authToken : string) : void	Adds a Product to a customer’s Basket with given basket and product id’s, and item count.

removeBasketItem	(basketId : string, productId : string, itemCount : uint, authToken : string) : void	Removes a Product from a customer's Basket with given basket and product id's, and item count.
clearBasketItem	(basketId : string, authToken : string) : void	Clears customer's basket of items.
defineDevice	(deviceId : string, name : string, aisle : string, type : enum, authToken : string) : Sensor	Creates a new device (Sensor or Appliance). "type" is either a Sensor type enum or Appliance type enum. Returns the device object.
showDevice	(storeId : string, deviceId : string) : void	Shows details of the device with given id in stdout.
createEvent	(deviceId : string, event : string, authToken : string) : void	Sends an event to device with given id.
createCommand	(applianceId : string, command : string, authToken : string) : void	Sends a command to Appliance with given id.

Store

The Store class represents an individual store serviced by the Modeler.

Properties

Property Name	**Type**	**Description**
storeId	string	A globally unique identifier assigned by provided input.
name	string	The name of the store.
address	string	The address of the store.

Associations

Association Name	**Type**	**Description**
customers	map<string : Customer>	The mapping of customer id's to customers currently in the store.
aisles	map<string : Aisle>	The mapping of aisle id's to aisles in the store.

inventories	map<string : Inventory>	The mapping of inventory id's to inventories of the store.
devices	map<string : Sensor>	The mapping of device id's to devices in the store.

Aisle

The Aisle class represents an individual aisle within a Store.

Properties

Property Name	Type	Description
number	string	Aisle number (unique within the store).
name	string	Name of aisle.
description	string	Description of aisle.
location	enum	Values: floor, storeroom.

Associations

Association Name	Type	Description
shelves	map<string, Shelf>	The mapping of shelf id's to shelves in an aisle.

Shelf

The Shelf class represents an individual shelf that's in an aisle.

Properties

Property Name	Type	Description
shelfId	string	Shelf id (unique within an aisle).
name	string	Name of shelf.
level	enum	Values: high, medium, low.
description	string	Description of shelf.

temperature	enum	Values: frozen, refrigerated, ambient, warm, hot; default is ambient.

Inventory

The Inventory class represents an individual inventory for a store. It defines the relationship between a product and its shelf location(s). The requirements state that the inventory must maintain the product count between zero and capacity inclusive which this class does.

Properties

Property Name	Type	Description
inventoryId	string	Inventory id (globally uniqe).
location	string	Location, store_id:aisle_id:shelf_id.
capacity	unsigned integer	Maximum amount of products allowed on shelf.
count	unsigned integer	Current amount of product on shelf.
productId	string	Product's id.

Product

The Product class represents an individual product item that the store can sell. Customers can add and remove items from their basket. The Inventory class defines relationships between Products and Shelves.

Properties

Property Name	Type	Description
productId	string	Product's id (globally unique).
name	string	Name of product.
description	string	Description of product.
size	string	Weight and/or volume of product.
category	string	Category of product, e.g., "dairy".
unitPrice	unsigned integer	Price of product in blockchain currency units.

temperature	enum	Values: frozen, refrigerated, ambient, warm, hot; default is ambient.

Customer

The Customer class represents a person who can shop at any store. Per the requirements, this class can differentiate between someone who is known/registered and unknown /a guest. Customers can also be either an adult or child. Known customers are recognized by all stores. Guests are not allowed to remove items from the store.

Properties

Property Name	Type	Description
customerId	string	Id of customer (globally unique).
firstName	string	First name of customer.
lastName	string	Last name of customer.
ageGroup	enum	Values: adult, child.
type	enum	Values: registered, guest.
emailAddress	string	Email address of customer.
blockchainAddress	string	Blockchain account used for billing the customer.
currentLocation	string	Store aisle the customer is currently in.
timeLastSeen	string	The time the customer was last seen by a device.

Basket

Baskets are assigned to customers upon entering a store. Customers can add, and remove its items.

Properties

Property Name	Type	Description
basketId	string	Id of basket (same as customer's id).
itemList	map<string, uint>	A list of products basket contains and their amount.

Sensor

The Sensor class represents an individual sensor in a store. Sensors are IoT devices in the store that can capture data and share it. They receive events, e.g., when the customer puts an item in their basket.

Properties

Property Name	Type	Description
deviceId	string	Sensor's id (globally unique).
name	string	Name of sensor.
location	string	Store aisle location of sensor.
type	enum	Values: microphone, camera.

Methods

Method Name	Signature	Description
receiveEvent	(event : String) : void	Prints an opaque string to stdout.

Appliance

The Appliance class represents an individual appliance in the store. Appliances are like sensors except they can be controlled and receive commands. The Appliance class extends the Sensor class.

Properties

Property Name	Type	Description
deviceId	string	Id of appliance (globally unique).
name	string	Name of appliance.
location	string	Store aisle location of sensor.
type	enum	Values: speaker, robot, turnstile.

Methods

Method Name	Signature	Description
receiveEvent	(event : string) : void	Prints an opaque string to stdout.

receiveCommand	(command : string) : void	Prints an opaque string to stdout.

CommandProcessor

The CommandProcessor is a utility class that per the requirements, supports a Command Line Interface (CLI) with a specific domain-specific language (DSL) for accessing the Store Model Service interface methods. The CLI commands can be in a script file or entered manually one by one. Commands and comments (prefaced with a '#') are printed to stdout.

Command syntax is specified as follows (from requirements):

Store Commands

```
# Define a store
define store <identifier> name <name> address <address>
# Show the details of a store, Print out the details of the store including the id, name, address,
active customers, aisles, inventory, sensors, and devices.
show store <identifier>
```

Aisle Commands

```
# Define an aisle within the store
define aisle <store_id>:<aisle_number> name <name> description <description> location (floor | store_room)
# Show the details of the aisle, including the name, description and list of shelves.
show aisle <store_id>[:<aisle_number>]
```

Shelf Commands

```
# Define a new shelf within the store
define shelf <store_id>:<aisle_number>:<shelf_id> name <name> level (high | medium | low) description <description> [temperature (frozen | refrigerated | ambient | warm | hot )]
# Show the details of the shelf including id, name, level, description and temperature
show shelf <store_id>[:<aisle_number>[:<shelf_id>]]
```

Inventory Commands

```
# Define a new inventory item within the store
define inventory <inventory_id> location <store_id>:<aisle_number>:<shelf_id> capacity <capacity> count <count> product <product_id>
# Show the details of the inventory
show inventory <inventory_id>
# Update the inventory count, count must >= 0 and <= capacity
update inventory <inventory_id> update_count <increment or decrement>
```

Product Commands

```
# Define a new product
define product <product_id> name <name> description <description> size <size> category <category> unit_price <unit_price> [temperature (frozen |
```

```
refrigerated | ambient | warm | hot )]
# Show the details of the product
show product <product_id>
```

Customer Commands

```
# Define a new customer
define customer <customer_id> first_name <first_name> last_name <last_name>
type (registered|guest) email_address <email> account <account_address>
# Update the location of a customer
update customer <customer_id> location <store:aisle>
# Show the details of the customer
show customer <customer_id>
```

Basket Commands

```
# Get basket_id associated with the customer, create new basket if the customer does
not
already have a basket associated.
get_customer_basket <customer_id>
# Add a product item to a basket
add_basket_item <basket_id> product <product_id> item_count <count>
# Remove a product item from a basket
remove_basket_item <basket_id> product <product_id> item_count <count>
# Clear the contents of the basket and remove the customer association
clear_basket <basket_id>
# Get the list of product items in the basket, include the product_id and count
Show basket_items <basket_id>
```

Sensor Commands

```
# Define device of type sensor
define device <device_id> name <name> type (microphone|camera) location
<store>:<aisle>
# Show device details
show device <device_id>
# Create a sensor event, this simulates a sensor event
create_event <device_id> event <event>
```

Note: Sensor events will be defined as part of the store controller module (i.e., assignment 3).

For now, treat as an opaque string.

Appliance Commands

```
# Define device of type appliance
define device <device_id> name <name> type (speaker | robot | turnstile)
location <store>:<aisle>
# Show device details
show device <device_id>
# Create an appliance event, this simulates a sensor event
create event <device_id> event <event_description>
# Send the appliance a command
create command <device_id> message <command>
```

Methods

Method Name	Signature	Description
processCommand	(command : string) : void	Processes commands manually. Throws CommandProcessorException on error.
processCommandFile	(commandFile : string) : void	Processes commands listed in a given commandFile. Throws CommandProcessorException on error.

ModelerException

The ModelerException is thrown by exceptions thrown from the Modeler. The exception includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed during exception occurrence.
reason	string	Reason for exception being thrown.

CommandProcessorException

The CommandProcessorException is thrown by exceptions thrown from the CommandProcessor. The action that caused the exception and the reason for it must be included. For commands read from a file, the file's line number where the exception occurred must also be included.

Properties

Property Name	Type	Description
action	string	Command performed during exception occurrence.
reason	string	Reason for exception being thrown.
lineNumber	integer	Line number of input file where the exception happened.

Implementation Details

The Store Model Service is a very stateful service, i.e., there's not much behavior involved with it. Objects need to be accessible and updateable per the requirements but not much more functionality is needed. The service's API is mostly a gateway to achieving these tasks and any external communication with the Store Model Service, e.g., by other modules of the 24X7 store, needs to use its public interface.

Nonetheless, most if not all assets should have a unique identifier (resepective of scope). Additionally, a product should be able to be on more than one shelf, so having more than one Inventory object with the same location and product id combination would be impractical. Furthermore, the customer's id could be used for the Basket class' basket id, and the basket can function as a virtual basket. In addition, the Store Controller (will be implemented later) will handle exceptions pertaining to inventory, basket, and shelf item accounting. For this module, item moving to and from both inventory and baskets can be thought of as separate activities.

Also, the requirements specify that some attributes can have default values such as for temperature in the Product and Shelf classes. To address this issue in Java, one could consider that two different methods can have the same name if they take in different parameters. Of note is that the interface diagram in this design document has two "defineProduct" and "defineShelf" methods.

Also of note is that many of the CLI command syntax commands use colon delimited strings as parameters which need to be parsed. In addition, one should consider reviewing the CLI command syntax thoroughly before implementing the service, e.g., for deciding how to scope objects and their methods and attributes. For instance, the "show inventory" command only accepts an inventory id and doesn't accept location information.

Finally, this design document uses maps extensively for storing and obtaining objects and items. This is because maps have constant-time look-up. They are convenient to use and highly recommended.

Exception Handling

Per the requirements, the following scenarios should warrant throwing exceptions:

- When a CLI command has invalid syntax.
- When a command would cause inventory product count to fall out of allowed range.
- Attempting to assign a non-registered customer with a basket.
- Attempting to assign a store a globally nonunique id.

In addition, for practicality, usability, and design purposes, the following scenarios should be considered for exception handling:

- When duplicate identifiers are submitted for creating new assets.
- When Inventory objects with the same location and product id combination are trying to be created.
- When searching for objects with a given id are not found or do not exist.

Testing

Testing is done through a test driver class called TestDriver with a “main” method as was done in the Ledger Service module. The main method calls the CommandProcessor class’s methods in order to process CLI commands manually (using the processCommand method) or through an input file (using the processCommandFile method) of choice. Like with the Ledger Service, the TestDriver class should be defined within a branched package that is called “com.cscie97.store.test”. For this design document, a command script is provided called “modeler.script”.

Risks

To be of practical use in business, the Store Model Service implementation should be altered to allow for state to be saved long term so that progress is not lost. The system could also benefit from added security such as encrypting customer information (which includes billing information), authenticating users and access to store assets including devices, and the like. The Authentication Service module of the Store 24X7 System hasn’t been implemented yet so these issues could be addressed then.