Model Service Design Document

Date: 9/15/2020
Author: Christopher Jones
Reviewer(s):

- Alejandro Chaparro
- Hidai Bar-Mor

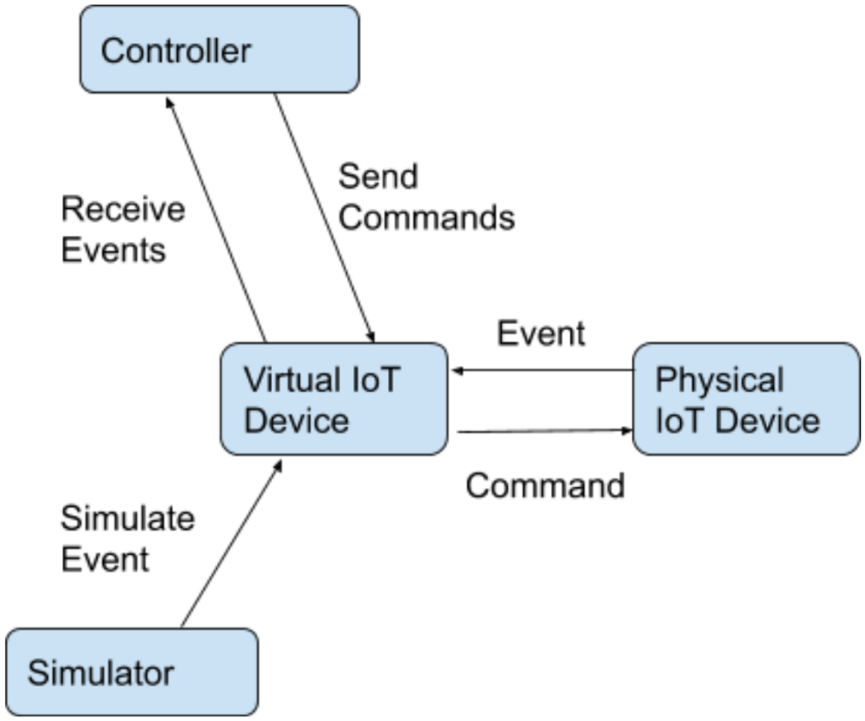
Introduction

This document provides the requirements for the Smart City Model Service.

Overview

The Model Service is responsible for maintaining the state of Internet of Things (IoT) devices within the Smart City. The IoT Devices (for which we will describe simply as devices moving forward) are designed to support the needs of city residents and visitors Every device contains sensors that are able to collect and share data. Sensors include cameras, microphones, CO2 sensors, thermometers, speakers, and GPS systems. All devices can be controlled through a set of commands that they receive from an external controller. The devices managed by the Model Service include Street Lights, Parking Meters, Street Signs, Information Kiosk, Robots, and Vehicles.

Physical IoT Devices are managed using Virtual IoT Devices that represent the state of the associated physical device. The Virtual IoT device provides a proxy to the physical device.
The following diagram shows the relationship between the virtual and physical IoT devices. It also shows how a Controller receives events and sends commands from and to the Physical IoT device via the Virtual IoT device.



For testing, a Simulator generates simulated events that appear to have come from the Physical IoT device. The Simulator will be used for testing the Smart City system.

Requirements

This section defines the requirements for the Smart City Model Service.

The Smart City Model Service is primarily responsible for managing the state of the City domain objects including:

- City
- People
    - Resident
    - Visitor
- Devices
    - Street Sign
    - Information Kiosk
    - Street Light
    - Robot
    - Parking Space
    - Vehicle
        - Bus
        - Car

Model Service Design Document
CSCI E-97 Assignment 2

City

The City is used to model a city instance. Note that the Smart City system is a cloud-based service and must be able to manage multiple Cities. A City has the following attributes:

- Globally unique identifier (e.g. city-1)
- Name (e.g. “Cambridge, MA”)
- Multiple people, either residents or visitors
- Multiple IoT Devices
- A blockchain account for receiving and sending money
- Location (latitude, longitude)
- Radius which specifies the area encompassed by the city

Person

Persons model the people that live in the city. A Person can be either a Resident or Visitor. Residents are well known persons, where visitors are anonymous. Both Residents and Visitors are assigned a unique person id. Attributes of Residents include:

- Globally unique ID
- Biometric ID
- Name of resident
- The phone number of the resident
- The Role of the resident (adult, child, or public administrator)
- Blockchain Account Address
- Location (latitude, longitude)

Devices

Devices are the internet connected components of the Smart City. All devices have the following attributes:

- A globally unique ID
- Location (latitude, longitude)
- Current status (ready, offline)
- Enabled (on/off)
- The latest event emitted from the device

All devices have the following input sensors:

- Microphone
- Camera
- Thermometer
- CO2 Meter

All devices have the following output sensors:

- Speaker
- GPS

The following section describes the different types of devices:

Street Sign

A street sign is an IoT device that provides information for vehicles. It is able to alter the text displayed on the sign. For example, it can dynamically adjust the speed limit, or warn about an accident ahead.

Information Kiosk

The Information Kiosk helps residents and visitors. It is able to interact with Persons, though speech and displaying images. For example, the Kiosk can display a map and help provide directions. The Kiosk can also support purchasing tickets for concerts and other events.

Street Light

The Street Light is a device for illuminating the city. The Street Light is able to adjust its brightness.

Robot

Robots act as public servants. Robots are mobile and can respond to commands from Residents and Visitors. For example, helping to carry groceries. They can also asset in emergencies, for example putting out a fire.

Parking Space

A Parking Space is an IoT device able to detect the presence of a vehicle. A parking space has an hourly rate which is charged to the account associated with the vehicle.

Vehicle

Vehicles are mobile IoT Devices that are used for giving rides to Residents and Visitors. Vehicles can be either a Bus or a Car. Vehicles have a maximum rider capacity. Both Cars and Busses are autonomous. Riding in a Bus or Car is free for Visitors, but requires a fee for Residents.

Bus

Buses are vehicles that can have a maximum rider capacity of 200, and a minimum ride capacity of 10.

Car

Cars are vehicles that can have a maximum ride capacity of 20, and a minimum ride capacity of 1.

Smart City Model Service

The Smart City Model Service provides a top-level Service interface for provisioning cities. It also supports controlling the City's IoT devices. Any external entity that wants to interact with the Smart City Model Service, must access it through the public API of the Model Service.

The Model Service provides a service interface for managing the state of the Cities.

The API supports commands for

- Defining the City configuration
- Showing the City configuration
- Updating the City configuration Creating/Simulating sensor events
- Sending command messages to IoT Devices Accessing IoT State and events
- Monitoring and supporting Residents and Visitors

All API methods should include an auth_token parameter that will be used later to support access control.

Command API

The Smart City Model Service supports a Command Line Interface (CLI) for configuring Cities and generating simulated sensor events. The commands can be listed in a file to provide a configuration script. The CLI should use the service interface to implement the commands.

Command Syntax

Model Commands

```
# Initialize Model / Return initial key
initialize model name <name> description <description> seed <seed>
[auth_token (token)]
```

City Commands

```
# Define a city
define city <city_id> name <name> account <address> lat <float> long
<float> radius <float> [auth_token (token)]

# Show the details of a city. Print out the details of the city
# including the id, name, account, location, people, and IoT devices.
show city <city_id> [auth_token (token)]
```

Device Commands

```
# Define a street sign
```

```
define street-sign <city_id>:<device_id> lat <float> long <float>
enabled (true|false) text <text> [auth_token (token)]

# update a street sign
update street-sign <city_id>:<device_id> [enabled (true|false)] [text
<text>] [auth_token (token)]

# Define an information kiosk
define info-kiosk <city_id>:<device_id> lat <float> long <float>
enabled (true|false) image <uri> [auth_token (token)]

# Update an information kiosk
update info-kiosk <city_id>:<device_id> [enabled (true|false)] [image
<uri>] [auth_token (token)]

# Define a street light
define street-light <city_id>:<device_id> lat <float> long <float>
enabled (true|false) brightness <int> [auth_token (token)]

# Update a street light
update street-light <city_id>:<device_id> [enabled (true|false)]
[brightness <int>] [auth_token (token)]

# Define a parking space
define parking-space <city_id>:<device_id> lat <float> long <float>
enabled (true|false) rate <int> [auth_token (token)]

# Update a parking space
update parking-space <city_id>:<device_id> [enabled (true|false)]
[rate <int>] [auth_token (token)]

# Define a robot
define robot <city_id>:<device_id> lat <float> long <float> enabled
(true|false) activity <string> [auth_token (token)]

# Update a robot
update robot <city_id>:<device_id> [lat <float> long <float>] [enabled
(true|false)] [activity <string>] [auth_token (token)]

# Define a vehicle
define vehicle <city_id>:<device_id> lat <float> long <float> enabled
(true|false) type (bus|car) activity <string> capacity <int> fee <int>
[auth_token (token)]

# Update a vehicle
```

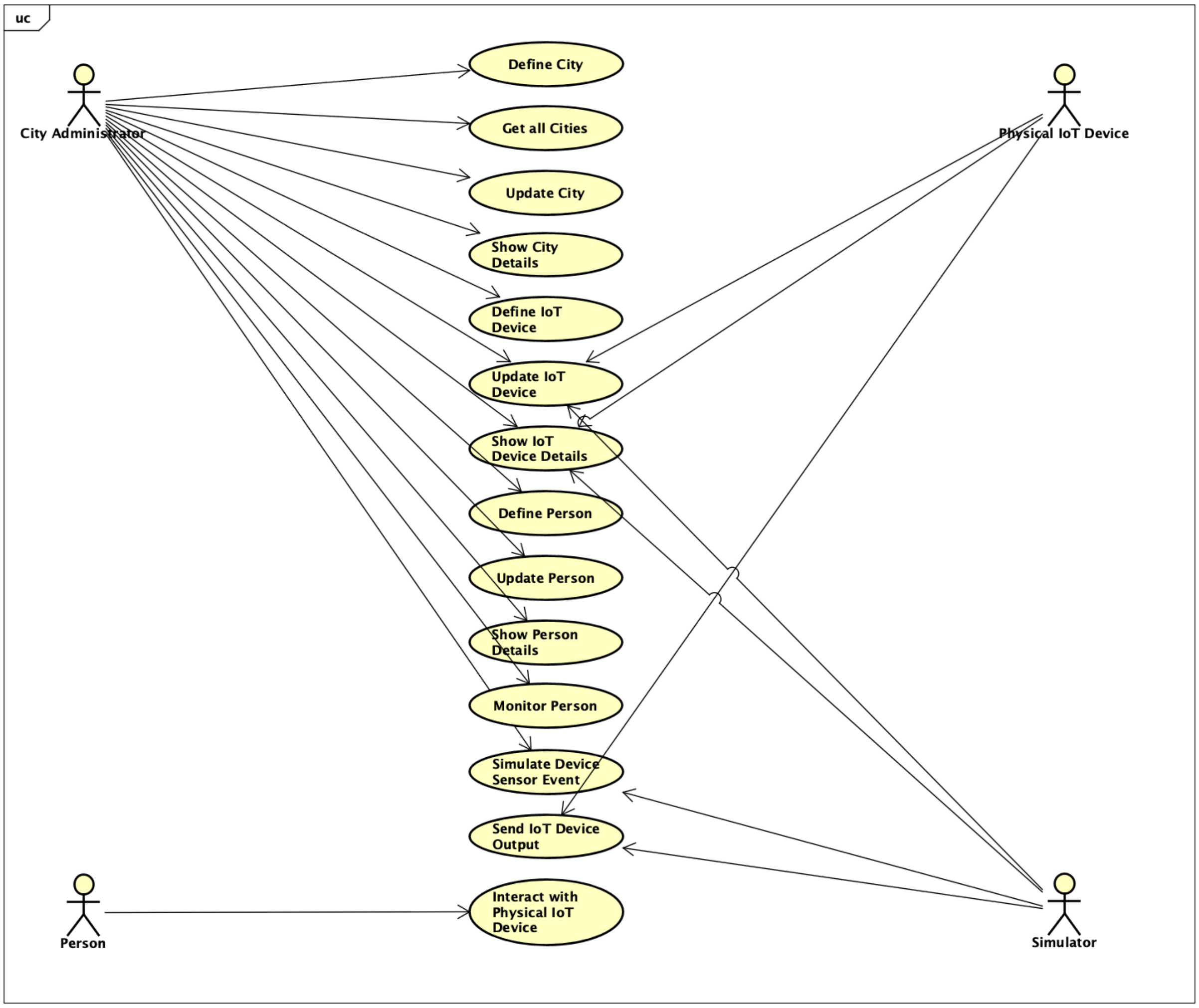
```
update vehicle <city_id>:<device_id> [lat <float> long <float>]
[enabled (true|false)] [activity <string>] [fee <int>] [auth_token
(token)]

# Show the details of a device, if device id is omitted, show details
# for all devices within the city
show device <city_id>[:<device_id>] [auth_token (token)]

# Simulate a device sensor event
create sensor-event <city_id>[:<device_id>] type
(microphone|camera|thermometer|co2meter) value <string> [subject
<person_id>] [auth_token (token)]

# Send a device output
create sensor-output <city_id>[:<device_id>] type (speaker) value
<string> [auth_token (token)]
```

Person Commands

```
# Define a new Resident
define resident <person_id> name <name> bio-metric <string> phone
<phone_number> role (adult|child|administrator) lat <lat> long <long>
account <account_address> [auth_token (token)]

# Update a Resident
update resident <person_id> [name <name>] [bio-metric <string>] [phone
<phone_number>] [role (adult|child|administrator)] [lat <lat> long
<long>] [account <account_address>] [auth_token (token)]

# Define a new Visitor
define visitor <person_id> bio-metric <string> lat <lat> long <long>
[auth_token (token)]

# Update a Visitor
update visitor <person_id> [bio-metric <string>] [lat <lat> long
<long>] [auth_token (token)]

# Show the details of the person
show person <person_id> [auth_token (token)]
```

Use Cases

The following Use Case diagram descripts the use cases supported by the Model System.



There are 4 types of actors:

- City Administrators
- Person
- Physical IoT Devices
- Simulator

City Administrators are responsible for configuring the smart city. This includes defining the city, provisioning the IoT devices, and setting up the identities of the residents.

Persons are the residents and guests that inhabit that city. Maximizing the person's experience in very important. Residents and visitors can interact with the various IoT devices and request services.

Physical IoT Devices monitor the city. For example, one type of IoT device, the Robot Public Servant, maintains the city, including cleaning the city, assisting people, and responding to emergencies. IoT devices are fully automated and also respond to requests from persons and the Smart City Controller.

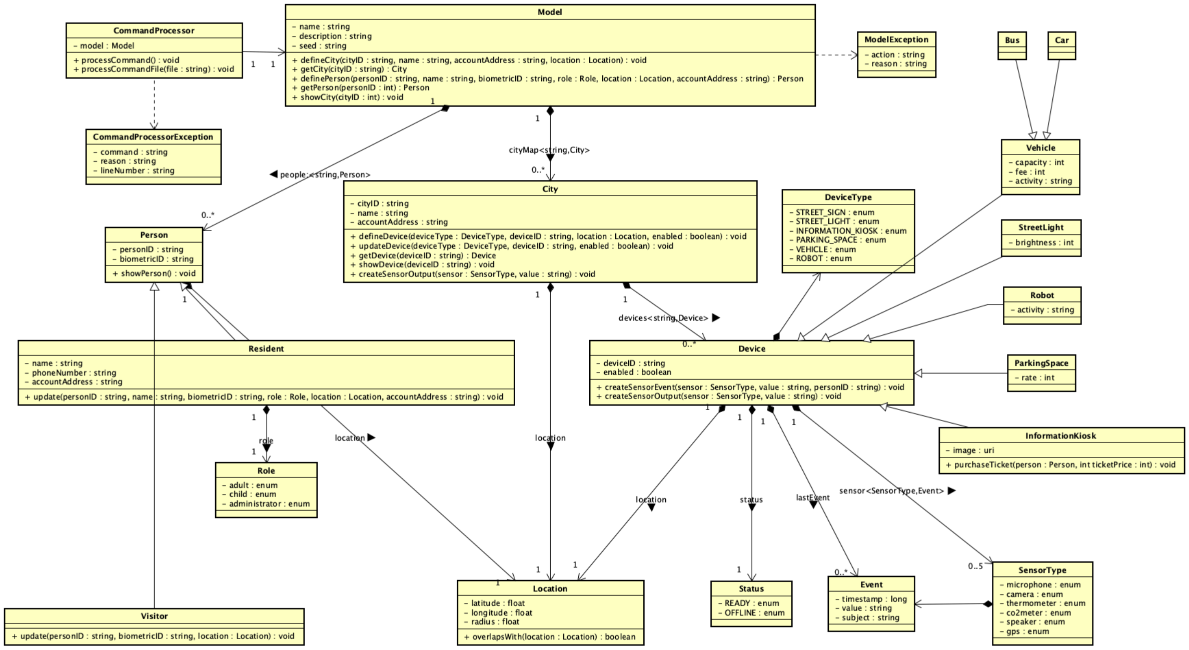
The **Simulator** supports testing the Smart City system by providing a source for the sensor events in place of using actual physical IoT Devices.

Implementation

The following defines the implementation details of the Model Service.

Class Diagram

The following class diagram defines the Model Service implementation classes contained within the package "cscie97.smartcity.model".


CommandProcessor
- model : Model
+ processCommand() : void
+ processCommandFile(file : string) : void
1
1
Model
- name : string
- description : string
- seed : string
+ defineCity(cityID : string, name : string, accountAddress : string, location : Location) : void
+ getCity(cityID : string) : City
+ definePerson(personID : string, name : string, biometricID : string, role : Role, location : Location, accountAddress : string) : Person
+ getPerson(personID : int) : Person
+ showCity(cityID : int) : void
ModelException
- action : string
- reason : string
Bus
Car
CommandProcessorException
- command : string
- reason : string
- lineNumber : string
1
1
cityMap<string,City>
0..*
◀ people:<string,Person>
0..*
Vehicle
- capacity : int
- fee : int
- activity : string
City
- cityID : string
- name : string
- accountAddress : string
+ defineDevice(deviceType : DeviceType, deviceID : string, location : Location, enabled : boolean) : void
+ updateDevice(deviceType : DeviceType, deviceID : string, enabled : boolean) : void
+ getDevice(deviceID : string) : Device
+ showDevice(deviceID : string) : void
+ createSensorOutput(sensor : SensorType, value : string) : void
DeviceType
- STREET_SIGN : enum
- STREET_LIGHT : enum
- INFORMATION_KIOSK : enum
- PARKING_SPACE : enum
- VEHICLE : enum
- ROBOT : enum
StreetLight
- brightness : int
Person
- personID : string
- biometricID : string
+ showPerson() : void
1
1
1
devices<string,Device> ▶
0..*
Robot
- activity : string
Resident
- name : string
- phoneNumber : string
- accountAddress : string
+ update(personID : string, name : string, biometricID : string, role : Role, location : Location, accountAddress : string) : void
Device
- deviceID : string
- enabled : boolean
+ createSensorEvent(sensor : SensorType, value : string, personID : string) : void
+ createSensorOutput(sensor : SensorType, value : string) : void
ParkingSpace
- rate : int
1
role
1
location ▶
location
1
1
1
1
InformationKiosk
- image : uri
+ purchaseTicket(person : Person, int ticketPrice : int) : void
Role
- adult : enum
- child : enum
- administrator : enum
location
status
lastEvent
sensor<SensorType,Event> ▶
1
1
1
1
1
0..*
0..5
Location
- latitude : float
- longitude : float
- radius : float
+ overlapsWith(location : Location) : boolean
Status
- READY : enum
- OFFLINE : enum
Event
- timestamp : long
- value : string
- subject : string
SensorType
- microphone : enum
- camera : enum
- thermometer : enum
- co2meter : enum
- speaker : enum
- gps : enum
Visitor
+ update(personID : string, biometricID : string, location : Location) : void

Class Dictionary

This section specifies the class dictionary for the Ledger System. The classes should be defined within the package "cscie97.smartcity.model".

CommandProcessor

The CommandProcessor is a utility class for feeding the Model a set of operations, using command syntax.

Properties

Property Name	Type	Description
model	Model	An instance of the Model class

Methods

Property Name	Type	Description
+ processCommand	(command:string):void	Process a single command. The output of the command is formatted and displayed to stdout. Throw a CommandProcessorException on error if any issues occur.
+ processCommandFile	(file:string):void	Process a set of commands provided within the given commandFile. Throw a CommandProcessorException on error. Otherwise, process commands directly in the interface.

CommandProcessorException

The CommandProcessorException is returned from the CommandProcessor methods in response to an error condition. The CommandProcessorException captures the command that was attempted and the reason for the failure. In the case where commands are read from a file, the line number of the command should be included in the exception.

Properties

Property Name	Type	Description
command	string	Command that was performed (e.g., “show person resident_1”)
reason	string	Reason for the exception (e.g. “cities cannot overlap”).
lineNumber	int	The line number of the command in the input file.

Model

Model Service Design Document
CSCI E-97 Assignment 2

The Model Service is primarily responsible for managing the state of the City domain objects. It manages the cites, their devices, and their inhabitants within the system. It also provides the API used by the clients of the Model Service. It contains a name, a description, and a unique seed that helps establish its identity.

Properties

Property Name	Type	Description
name	string	The name of the model (i.e. smartcitymodel)
description	string	The description of the model
seed	string	The seed used to generate api keys by roles in future projects

Methods

Property Name	Type	Description
defineCity	(cityID:string, name:string, accountAddress: string, location:Location):void	Create and add a new city to the model if it doesn't already exist or overlap with others, otherwise, throw a ModelException.
getCity	(cityID:string):City	Get a city within the model if it exists, otherwise, throw a ModelException.
definePerson	(personID:string, name:string, biometricID:string, role:Role, location:Location, accountAddress:string):Person	Create and add a new Resident/Visitor to the model if it doesn't already exist, otherwise, throw a ModelException.
getPerson	(personID:string):Person	Get a resident/visitor within the model if it exists, otherwise, throw a ModelException.
showCity	(cityID):void	Display details about the city and its residents

Associations

Association	Type	Description
cityMap	Map<string,Person>	A map of all the cities within the model
people	Map<string,City>	A map of all people within the model

ModelException

The Model Exception is returned from the Model API methods in response to an error condition. The Model Exception captures the action that was attempted and the reason for the failure.

Properties

Property Name	Type	Description
action	string	The action that was attempted that brought about the exception
reason	string	The reason that the action caused an error

City

The City class represents an individual city within the Model Service. It has a predefined location with its own boundaries, its own subset of residents/visitors, and its own devices set up to assist them. City boundaries cannot be overlapping.

Properties

Property Name	Type	Description
cityID	string	A unique ID identifying the city
name	string	The name of the city
accountAddress	string	The city's blockchain account address

Methods

Property Name	Type	Description
defineDevice	(deviceType:DeviceType, deviceID:string, location:Location,enabled:Boolean):void	Create a new device of deviceType if it is valid and does not already exist in the device list.
updateDevice	(deviceType:DeviceType, deviceID:string, enabled:Boolean):void	Update an existing device of deviceType.

getDevice	(deviceID:string):Device	Get a specific device from the device list if it exists.
showDevice	(deviceID:string):void	Display details about a device if it exists.
createSensorOutput	(sensor:SensorType, value:string):void	Broadcast a message to all devices within a city.

Associations

Association	Type	Description
location	Location	The general location of the city given by latitude, longitude, and the amount of overall space it takes up
devices	Map<string, Device>	A list of all the devices associated with that city

Location

The Location class represents a specified point or space within a City. Cities have locations that embody an entire space, while people and devices only embody points within an existing City. City location boundaries cannot overlap.

Properties

Property Name	Type	Description
latitude	float	The latitude of the location
longitude	float	The longitude of the location
radius	float	The radius of the location

Methods

Property Name	Type	Description
overlapsWith	(L:Location):boolean	Given a location, check whether or not it and this location overlaps with each other

Device

IoT Devices are the internet connected components of the Smart City. All IoT devices have specified locations, device IDs, a status (ready/offline), an enabled status (on/off), a set of predefined sensors associated with it (microphone, CO2 meter, camera, and thermometer), and its most recent event emitted by one of those sensors.

Properties

Property Name	Type	Description
deviceID	string	A unique identifier for the device
enabled	boolean	An identifier of whether or not the device is enabled or not

Methods

Property Name	Type	Description
createSensorEvent	(sensor:SensorType, value:string, personID:string):void	Create a sensor event and update the most recent event sent
createSensorOutput	(sensor:SensorType, value:string):void	Create a sensor output event and update the most recent event sent

Associations

Association	Type	Description
location	Location	The general location of the city given by latitude, longitude, and the amount of overall space it takes up
status	Status	A special identifier of whether the device is ready or offline
lastEvent	Event	The most recently processed event. If no event was processed, it is null.

DeviceType

The device type is an identifier for the different types of devices that can be found without a city.

Properties

Property Name	Type	Description

STREET_SIGN	enum	A device identifier for street signs
STREET_LIGHT	enum	A device identifier for street lights
INFORMATION_KIOSK	enum	A device identifier for information kiosks
PARKING_SPACE	enum	A device identifier for a parking space
VEHICLE	enum	A device identifier for a vehicle
ROBOT	enum	A device identifier for a robot

InformationKiosk

The Information Kiosk helps residents and visitors. It is able to interact with Persons, though speech and displaying images. For example, the Kiosk can display a map and help provide directions. The Kiosk can also support purchasing tickets for concerts and other events.

Properties

Property Name	Type	Description
image	uri	An info-graphic to be displayed by the device

Methods

Property Name	Type	Description
purchaseTicket	(person:Person, int ticketPrice):void	Help someone purchase a ticket to an event

ParkingSpace

A Parking Space is an IoT device able to detect the presence of a vehicle. A parking space has an hourly rate which is charged to the account associated with the vehicle.

Properties

Property Name	Type	Description
rate	int	The cost of the parking space per hour

Robot

Robots act as public servants. Robots are mobile and can respond to commands from Residents and Visitors. For example, helping to carry groceries. They can also asset in emergencies, for example putting out a fire.

Properties

Property Name	Type	Description
activity	string	The activity assigned to this robot (i.e. “take out the trash”)

StreetLight

The Street Light is an IoT device for illuminating the city. The Street Light is able to adjust its brightness.

Properties

Property Name	Type	Description
brightness	int	The brightness setting of the light

StreetSign

A street sign is an IoT device that provides information for vehicles. It is able to alter the text displayed on the sign. For example, it can dynamically adjust the speed limit, or warn about an accident ahead.

Properties

Property Name	Type	Description
text	string	The text displayed on the street sign

Vehicle

Vehicles are mobile IoT Devices that are used for giving rides to Residents and Visitors. Vehicles can be either a Bus or a Car. Vehicles have a maximum rider capacity. Both Cars and Busses are autonomous. Riding in a Bus or Car is free for Visitors, but requires a fee for Residents.

Properties

Property Name	Type	Description

capacity	int	The maximum number of riders for this vehicle
fee	int	The cost for residents to ride this vehicle
activity	string	The task the vehicle is performing (i.e. “Picking up people”)

Bus

The Bus class is a subclass of the Vehicle class. Their predefined capacity must fall between 10 and 200.

Car

The Car class is a subclass of the Vehicle class. Their predefined capacity must fall between 1 and 20.

SensorType

The Sensor class generates events that are processed by the Virtual IoT Devices. There are 4 types of sensors, namely microphones, CO2 meters, cameras, and thermometers. Microphones are able to convert speech to text, and both the microphone and camera sensors use AI to automatically identify the subject person. The CO2 Sensor may generate the events measuring the total "parts per million" (denoted "ppm", and used to measure the concentration of a contaminant in soils and sediments), similarly, the thermometer may report the current ambient temperature, or the temperature of an individual person. There is additionally, a speaker sensor to help broadcast events to the devices.

Properties

Property Name	**Type**	**Description**
microphone	enum	A microphone sensor intended to process vocal interactions
camera	enum	A camera sensor intended to process visual interactions
thermometer	enum	A thermometer sensor intended to process thermal fluctuations
co2meter	enum	A CO2 meter sensor intended to process air quality changes

speaker	enum	A speaker sensor intended to broadcast events out to devices
GPS	enum	A GPS system intended to determine the current location of the device

Event

The Event class represents the individual events to be processed by a Sensor. Events have a type, action, and an optional subject.

Properties

Property Name	Type	Description
timestamp	long	An indicator of the time in which an event was processed
value	string	The general value of the event
subject	string	The subject or target the event was applied to

Status

The Status class is an enum class storing the values READY and OFFLINE to indicate the status of each of the individual devices. By default, they are all offline, but can be switched to ready on update.

Properties

Property Name	Type	Description
READY	enum	An indicator that a device is ready to use
OFFLINE	enum	An indicator that a device is offline

Person

The Person class represents the individual people within the a given City. A Person can be either a Resident or Visitor. Residents are well known persons, where visitors are anonymous. Both Residents and Visitors are assigned a unique person id.

Properties

Property Name	Type	Description
personID	string	The unique ID of the person
biometricID	string	The unique biometricID of the person
location	Location	The location that the person is currently located at

Methods

Property Name	Type	Description
showPerson	():void	Display the details of the resident/visitor

Resident

The Resident class is a subclass of the Person class. In addition to the above requirements, a Resident has a known name, phone number, and account within the city. Additionally, they can be identified as an adult, child, or public administrator. Children must have the same account address as an existing adult or public administrator, and a public administrator must have the same account as the master account holder.

Properties

Property Name	Type	Description
name	string	The name of the resident
phoneNumber	string	The phone number of the resident
accountAddress	string	
role	Role	

Methods

Property Name	Type	Description
update	(personID:string, name:string, biometricID:string, role:Role, location:Location, accountAddress:string)	Update the resident attributes

Visitor

The Visitor is a subclass of the Person class, and is any Person that isn't a Resident within that City. They can interact with the IoT Devices in the same capacity as Residents. However, their fees to use them are waived.

Methods

Property Name	Type	Description
update	(personID:string, biometricID:string, location:Location)	Update the visitor attributes

Role

The role of a resident identifies who they are in reference to the others. This can potentially be used to construct relationships in future iterations of this project.

Properties

Property Name	Type	Description
adult	enum	An identifier of an adult
child	enum	An identifier of a child
administrator	enum	An identifier of an administrator

Implementation Details

The Model application operates as the overarching management system for the City, the People, and the Devices. While the 3 are disjoint properties, the Model Services allows them to operate as a unit all while not having to worry about any form of shared state.

Some additional modifications made onto the original design included:

- providing the Model class a seed parameter to utilize for API key generation
- combining the latitude, longitude, and radius variables into the Location class
  - this can be utilized in methods that apply to all City, Device, and People
- including timestamps within the Event class in the event that important information is superseded by something less important
- initializing the Model class such that role support can be integrated from the very beginning

All of these are minor changes that fulfill the provided requirements all while bringing simplicity to future integrators.

Exception Handling

In this project there are two primary exception handling methods. The exceptions handled by the CommandProcessorException class, handle any and all events that are the result of the CommandProcessor. This is done in an effort to hide potentially security threatening information brought on by internal errors from getting to the user. The exceptions handled by the ModelException class, handle any and all errors occurring exclusively on the Model Service. The exceptions are to assist in troubleshooting the same internal issues discussed above.

Testing

As of now, the tests are managed within the TestDriver class that can perform both file-based or command-based processing based on whether or not you have an input script. This class can be found within the package **cscie97.smartcity.test**, and can be run by the following commands:

- Run the following command to build the code:
- javac cscie97/smartcity/model/*.java cscie97/smartcity/test/*.java
- Run the following command to test:
- java -cp . cscie97.smartcity.test.TestDriver smartcity_model.script

Risks

A lot of the information passed around in this application may bring about security concerns down the road. Ideally, while getters and setters were set for every variable, they should be available for others. Also displaying the person, device, and city IDs, should only be displayed to particular individuals, to prevent fraud.