Smart City Model Service Requirements

Brett Bloethner

Introduction

This document provides system architecture details for the Smart City Software System's Smart City Model Service. In this document you will find use case diagrams, class diagrams and implementation details that will help outline and explain the Smart City Model Service design.

Table of Contents

Overview

This Smart City Model Service is one of the four core microservices that makes up the Smart City System Architecture. The model service allows administrators to manage three key physical components that make up a smart city. These three components are the city, residents of the city and connected IoT devices in the city. Specifically, the model service allows for the creation and updating of city, resident and IoT device parameters as well as the event triggering of those same IoT devices.

Requirements

The Smart City Model Service is responsible for the creation and management of all of the "things" involved in the smart city, including the city itself. Below are outlines of each object created and managed by the smart city model service.

City

The Smart City Model service is capable of managing multiple cities. It can create cities and update cities. A city contains a name and cityId for identification as well as a location. Also included in the city is people and devices. People and devices can be located within the city but management of devices is done within the scope of the city while people are managed on a global scale. Cities also have a blockchain account address which can be used by people and devices to conduct business transactions.

Person

A person models any individual inside the city. There are two types of people, Residents and Visitors. Both residents and visitors have a personId, biometricId and location. Visitors are effectively anonymous while residents include a few more data parameters. In addition to personId, biometricId and location, residents also have a name, phone number, blockchain address, and role. Residents can have roles of "adult," "child," or "public administrator."

Device
A device represents any IoT device that resides in a city managed by the Smart City Model Service. IoT devices can be a street sign, information kiosk, street light, robot, parking space or vehicle. Each IoT device includes parameters for deviceId, location, status, enabled, blockchain address, and last sensor event. Each device also has sensors/inputs which generate events.

- **Street Sign:** Street signs can show vehicles information in the form of text. This text can be changed to provide whatever information may be necessary to nearby drivers and people.
- **Information Kiosk:** Information kiosks provide valuable information to residents in the form of interaction with residents using speech and images. These kiosks can also support other features like ticket sales. They include an image URI for what they're currently displaying.
- **Street Light:** A street light can be fine-tuned with an adjustable brightness setting.
- **Robot:** Robots help persons around the city. They can respond to commands to help with menial tasks like carrying things and they can also assist in emergencies.
- **Parking Space:** Parking spaces can detect when a car is present and charge the person parking with whatever the current hourly rate is.
- **Vehicle:** A vehicle can be either a car or a bus and both provide rides to visitors and residents. Both are also autonomous. Visitors ride in these vehicles for free while residents must pay a fee and each vehicle has a specified capacity of riders.

Device Sensors & Events
Each IoT Devices includes sensors to measure sound, imagery, temperature and CO2. These sensors emit events which include parameters for the type of event, the events action and the events subject which is an optional parameter. Each sensor also has the capability of communicating with persons by outputting sound.

- **Microphone:** the microphone automatically dictates phrases and events are constructed with a string version of what the phrase said was.
- **Camera:** the camera automatically analyzes visuals and will provide a string description of the event it witnessed, such as "person walking" or "car detected"
- **Thermometer:** the thermometer provides a number reading but is capable of reading ambient temperatures or the temperature of nearby people and devices
- **CO2 Meter:** the CO2 meter simply generates a reading of the current ambient CO2 level.

Controlling the Smart City Model Service
The Smart City Model service serves includes a public API. Through the Smart City Model Service API an actor can provision, view or change a city, person or IoT device. Devices and Persons can also be queried based on their range within a cities radius. Actors can also simulate sensor events, access IoT device's events and send commands to IoT devices. Monitoring and support of people in the city is also done through the Smart City Model Service API. Auth tokens

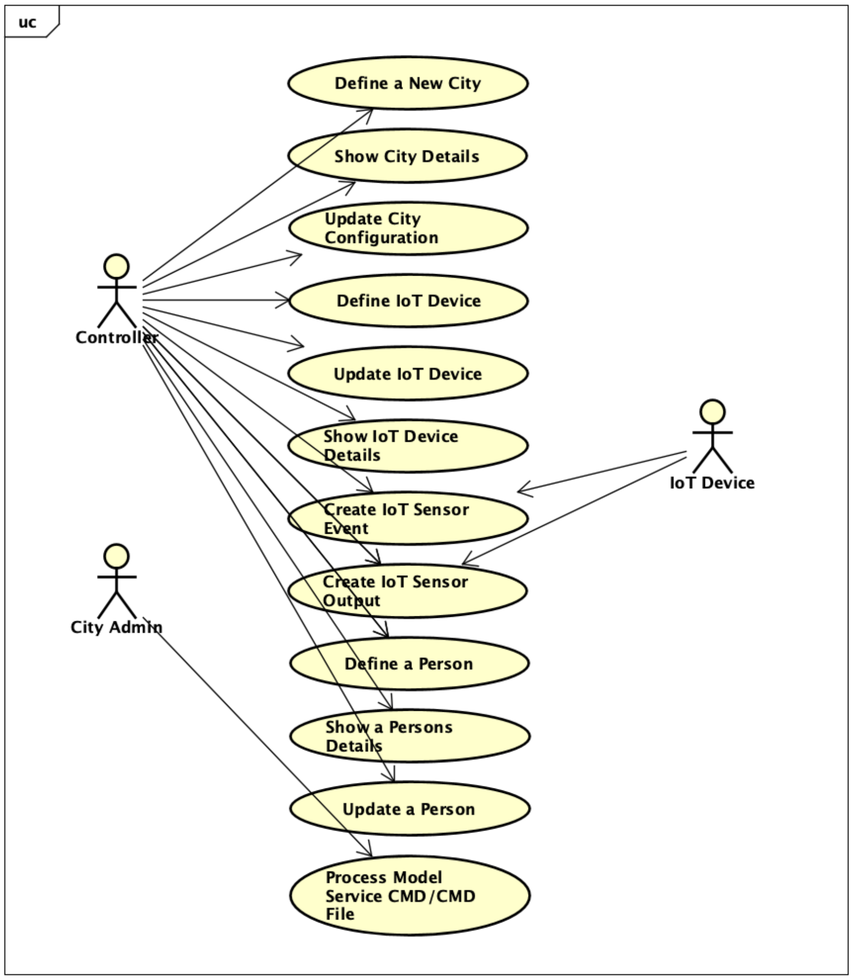
are used in each API call to provide access control mechanisms for security (but will be implemented at a later date). This API can be accessed programmatically via the Java Package or via the Command API which provides a CLI that enables full control of all the features provided by the Smart City Model Service.

Use Cases

Actors

- **Controller:** The controller of the smart city system that coordinates how the different components of the smart city system interact with each other
- **City Admin:** Administrators who manage the smart city
- **IoT Device:** Devices such as Vehicles and Information Kiosks which interface with other smart city components

Below is the use case diagram which outlines what the expected actors are and what abilities they'll have in the application.



Define a New City

- Verify all the required arguments are present
- Verify that the citys id is unique and create a Location object for it
- Return an error if there is a conflict with the configuration of a different city

- Create the city and initialize its device storage
- Store the new city in the Smart City Model Service

Show City Details/Configuration
- Get the city from the Smart City Model Service
- Get the devices in the area of the city
- Get the persons in the area of the city

Update a City’s Configuration
- Get the city from the Smart City Model Service city hash map, or error if doesn’t exits
- Determine what parameters were provided in the args for update
- Update the city with the new values from the command args

Define an IoT Device
- Verify all the required arguments are present for the type of device in the command
- Verify no device exists with the same deviceId
- Get the city that the device should be registered with
- Create a Location object using the location data provided (radius is defaulted to 0.0)
- Create the Device object using the IoT Device child class type from the command (street light, street sign, information kiosk, robot, vehicle, parking space)
- Store the device in the device hash map In the city the device is registered with
- Store the device in the global device hash map in the Smart City Model Service

Update an IoT Device
- Get (by city id and device id) the IoT device to be updated
- Create a copy of the IoT device in its current state
- Determine what parameters are to be updated, update those parameters on the copied device
- Replace (in the citys device hash map) the old device with the new device
- Replace (in the Smart City Model Service global device hash map) the old device with the new device

Show IoT Device Details/Configuration
- Get the city from the Smart City Model Service
- Get, from the city, the device from the citys device hash map

Simulate IoT Device Sensor Event
- Get the city from the Smart City Model Service
- Get, from the city, the device from the citys device hash map
- Validate the sensor event type argument used in the command
- Create a device sensor event object
- One the device object, pass the device event into the register device event method
- Handle the event accordingly (right now its just logging out the event)

Send IoT Device Sensor Output
- Get the city from the Smart City Model Service, or error if it doesn’t exist
- Get, from the city, the device from the citys device hash map, or error if doesn’t exist
- Validate the sensor output type argument used in the command
- Create a device sensor output object
- One the device object, pass the device sensor output into the register device output method
- Handle the output accordingly (right now its just logging out the output)

Define a Person
- Verify all the required arguments are present for the type of person in the command (visitor or resident)
- Determine if there is already a person with the same person id, if there is then return an error
- Create the new Visitor or Resident Person object, depending on the command
- Store that new person in the Smart City Model Service global person hash map

Update a Person
- Get the current person, by person id, from the global person hash map in the Smart City Model Service

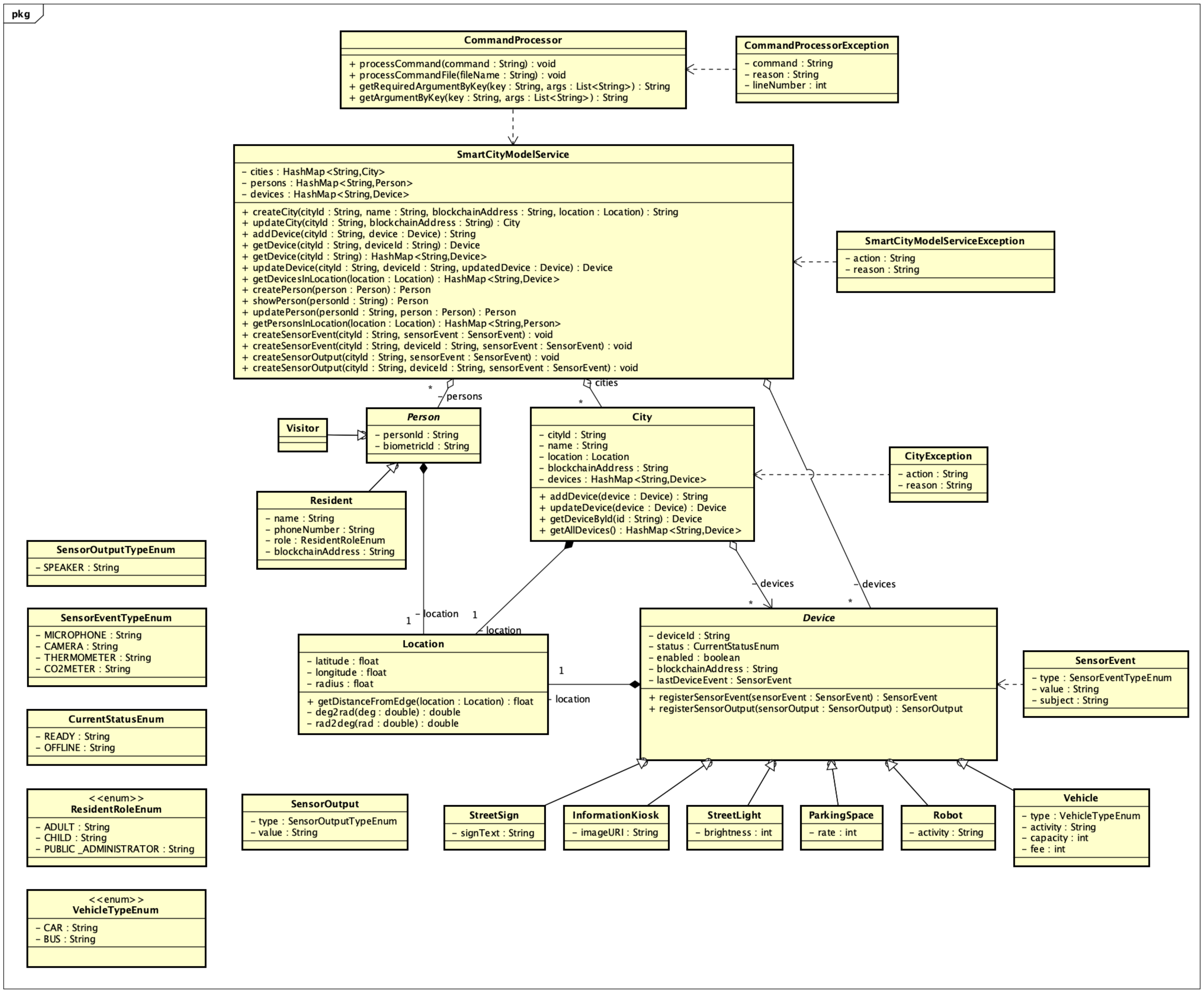
- Create a copy of the current person
- Determine what the person type is and what parameters are provided in the args to be updated
- Update the parameters on the copy of the person
- Replace (in the Smart City Model Service persons hash map) the old person with the new person

Show Person Details/Configuration

- Get the person, by person id, from the global person hash map in the Smart City Model Service
- Return an error if the person does not exist

Class Diagrams

The following class diagram defines the Smart City Model Service implementation classes contained within the package "com.cscie97.smartcity.model". Below is the class diagram outlining the design and relation of all the classes in the Smart City Model Service.


pkg
CommandProcessor
+ processCommand(command : String) : void
+ processCommandFile(fileName : String) : void
+ getRequiredArgumentByKey(key : String, args : List<String>) : String
+ getArgumentByKey(key : String, args : List<String>) : String
CommandProcessorException
- command : String
- reason : String
- lineNumber : int
SmartCityModelService
- cities : HashMap<String,City>
- persons : HashMap<String,Person>
- devices : HashMap<String,Device>
+ createCity(cityId : String, name : String, blockchainAddress : String, location : Location) : String
+ updateCity(cityId : String, blockchainAddress : String) : City
+ addDevice(cityId : String, device : Device) : String
+ getDevice(cityId : String, deviceId : String) : Device
+ getDevice(cityId : String) : HashMap<String,Device>
+ updateDevice(cityId : String, deviceId : String, updatedDevice : Device) : Device
+ getDevicesInLocation(location : Location) : HashMap<String,Device>
+ createPerson(person : Person) : Person
+ showPerson(personId : String) : Person
+ updatePerson(personId : String, person : Person) : Person
+ getPersonsInLocation(location : Location) : HashMap<String,Person>
+ createSensorEvent(cityId : String, sensorEvent : SensorEvent) : void
+ createSensorEvent(cityId : String, deviceId : String, sensorEvent : SensorEvent) : void
+ createSensorOutput(cityId : String, sensorEvent : SensorEvent) : void
+ createSensorOutput(cityId : String, deviceId : String, sensorEvent : SensorEvent) : void
SmartCityModelServiceException
- action : String
- reason : String
* - persons
cities *
Visitor
Person
- personId : String
- biometricId : String
City
- cityId : String
- name : String
- location : Location
- blockchainAddress : String
- devices : HashMap<String,Device>
+ addDevice(device : Device) : String
+ updateDevice(device : Device) : Device
+ getDeviceById(id : String) : Device
+ getAllDevices() : HashMap<String,Device>
CityException
- action : String
- reason : String
Resident
- name : String
- phoneNumber : String
- role : ResidentRoleEnum
- blockchainAddress : String
SensorOutputTypeEnum
- SPEAKER : String
SensorEventTypeEnum
- MICROPHONE : String
- CAMERA : String
- THERMOMETER : String
- CO2METER : String
CurrentStatusEnum
- READY : String
- OFFLINE : String
<<enum>>
ResidentRoleEnum
- ADULT : String
- CHILD : String
- PUBLIC_ADMINISTRATOR : String
<<enum>>
VehicleTypeEnum
- CAR : String
- BUS : String
1 - location
1 - location
- devices *
- devices *
Location
- latitude : float
- longitude : float
- radius : float
+ getDistanceFromEdge(location : Location) : float
- deg2rad(deg : double) : double
- rad2deg(rad : double) : double
1 - location
Device
- deviceId : String
- status : CurrentStatusEnum
- enabled : boolean
- blockchainAddress : String
- lastDeviceEvent : SensorEvent
+ registerSensorEvent(sensorEvent : SensorEvent) : SensorEvent
+ registerSensorOutput(sensorOutput : SensorOutput) : SensorOutput
SensorEvent
- type : SensorEventTypeEnum
- value : String
- subject : String
SensorOutput
- type : SensorOutputTypeEnum
- value : String
StreetSign
- signText : String
InformationKiosk
- imageURI : String
StreetLight
- brightness : int
ParkingSpace
- rate : int
Robot
- activity : String
Vehicle
- type : VehicleTypeEnum
- activity : String
- capacity : int
- fee : int

Class Dictionary

This section specifies the class dictionary for the Smart City Model Service. The classes should be defined within the package "com.cscie97.smartcity.model".

City

The city class represents a city and is responsible for storing the configuration of a city as well as the devices registered to that city.

Methods

Method Name	Signature	Description
addDevice	(Device device) String deviceId	Add a device to the citys devices hashmap
updateDevice	(Device device) Device device	Replace a device in the citys devices hashmap
getDeviceById	(String deviceId) Device device	Get a device from the citys devices hashmap
getAllDevices	() HashMap<String, Device>	Get all devices from the citys devices hashmap

Properties

Property Name	Type	Description
cityId	String	Globally unique id
name	String	Friendly name for the city
blockchainAddress	String	Address of the citys account

Associations

Association Name	Type	Description
location	Location	location and radius of the city
devices	HashMap<String, Device>	devices registered to this city

Command Processor

The command processor is responsible for reading processing commands from the command line or reading a command file. It filters on the numerous arguments provided by the user and determines what command should be run against the Smart City Model Service API. The Command Processor is designed to the spec in Assignment 1.

<u>The Commands and syntax can be found in the assignment documents.</u>

Methods

Method Name	Signature	Description
processCommand	(command:string):void	Process a single command.
processCommandFile	(commandFile:string):void	Process a set of commands provided within the given commandFile.

Command Processor Exception

The command processor exception is a multipurpose exception class used by the command processor object for reporting its exceptions

Properties

Property Name	Type	Description
command	String	Command that was performed
reason	String	Reason for the exception
lineNumber	String	The line number of the command in the input file

Device

The device class represent devices at a high level and contains parameters that each device type uses in its configuration. Device is an abstract class and each IoT device type is a child of it.

Methods

Method Name	Signature	Description
registerSensorEvent	(SensorEvent:sensorEvent): SensorEvent:sensorEvent	Handles an IoT sensor event in the context of the device instance. Sets last device event accordingly.
registerSensorOutput	(SensorOutput:sensorOutput): void	Handles an IoT sensor output such as a speaker dialog output.

Properties

Property Name	Type	Description
deviceId	String	Globally unique ID
blockchainAddress	String	Blockchain account address of the device
status	CurrentStatusEnum	Current status of the device
enabled	Boolean	Whether or not the device is enabled

Associations

Association Name	Type	Description
lastDeviceEvent	SensorEvent	The last Sensor Event that this device encountered
location	Location	The location of the device (radius is default, 0.0)

Information Kiosk *extends Device*

The information kiosk class is a child of *Device* and implements additional parameters that are specific to what an information kiosk might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
imageUri	String	The URI location of the image currently displayed by the kiosk

Location

The location class is implemented by objects in the smart city and stores the latitude, longitude and radius of any point/area. It can also tell you where it is in relation to another provided Location object. This helps fulfill the requirement of being able to get Persons and Device in a Citys specific radius.

Methods

Method Name	Signature	Description

getDistanceFromEdge	(Location:location):Float	Returns the distance from the edge of this location to the edge of the location passed in.
deg2Rad	(double:deg):double	Convert degrees to radians
rad2Deg	(double:rad):double	Convert radians to degrees

Properties

Property Name	Type	Description
latitude	Float	Latitude of the location
longitude	Float	Longitude of the location
radius	Float	Radius of the location (0.0 for devices and people)

Parking Space *extends Device*

The parking space class is a child of *Device* and implements additional parameters that are specific to what a parking space might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
rate	Int	The current pricing to use the parking space

Person

The person class represents persons at a high level and contains parameters that each person type uses in its configuration. Person is an abstract class. Visitor and Resident are a child of it.

Properties

Property Name	Type	Description
personId	String	Globally unique ID for this person
biometricId	String	Identifier used for authentication

Associations

Association Name	Type	Description
location	Location	The current location of the person

Resident *extends Person*

Resident inherits from the *Person* class and represents a resident in the smart city model service. It is not associated to a city but can be queried so that the city its currently in can be determined. The class includes personal information beyond what the *Person* class provides. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
name	String	The name of the resident
phoneNumber	String	The phone number of the resident
blockchainAddress	String	The blockchain account address of the resident
role	ResidentRoleEnum	The role of the resident

Robot *extends Device*

The robot class is a child of *Device* and implements additional parameters that are specific to what a robot might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
activity	String	Description of the current activity being conducted by the robot

Sensor Event

The Sensor Event class describes the occurrence of an event detected by sensors on an IoT *Device* object.

Properties

Property Name	Type	Description
type	SensorEventTypeEnum	The type of event the sensor detects
value	String	The value read by the sensor
subject	String	The subject read be the sensor (eg a person)

Sensor Output

The Sensor Output class describes the occurrence of output (eg speaker dialog output) executed by sensors on an IoT *Device* object.

Properties

Property Name	Type	Description
type	SensorOutputTypeEnum	The type of output being executed by the sensor
value	String	The information being output by the sensor

Smart City Model Service

The Smart City Model Service class manages everything having to do with models in the smart city and is the API by which models are managed by other systems in the smart city. It stores citys and their configurations in a hash map. To help with location monitoring requirement, it also stores global hash maps of all devices and all persons under supervision of the entire smart city service. The Smart City Model Service helps to fulfill many of the CRUD like features in the requirements for Devices, Persons and Citys.

Methods

Method Name	Signature	Description
createCity	(String:cityId, String:name, String:blockchainAddress,Location location) :String:cityId	Creates a new city and stores it in the cities hash map
updateCity	(String:cityId,String:blockchainAddress) :City city	Update the configuration for the city by cityId
getCityById	(String:cityId):City:city	Get the city by id from the cities hash map

addDevice	(String:cityId, Device:device)String:deviceId	Add a device to a city and to the global devices hash map
getDevice	(String:cityId,String:deviceId) :Device:device	Get a device from a city
getDevice	(String:cityId):Device:device	Get all of a citys devices
updateDevice	(String:cityId,String:DeviceId,Device:device): Device:device	Update/replace a device in a city and in the global cities hash map
getDevicesInLocation	(Location:location):HashMap<String, Device>	Get all of the devices in a geographic location
createPerson	(Person:person):Person	Create and add a person to the global persons hash map
showPerson	(String:personId):Person	Get, by id, a person from the global persons hash map
updatePerson	(String:personId,Person:person):Person	Update/replace a person, by id, in the global person hash map
getPersonsInLocatioon	(Location:location):HashMap<String,Persons>	Get all the persons in a geographic area
createSensorEvent	(String:cityId, String:deviceId, SensorEvent sensorEvent): void	Create/simulate a sensor event for a specific device in a city
createSensorEvent	(String:cityId, SensorEvent: sensorEvent):void	Create/simulate a sensor event for all devices in a city
createSensorOutput	(String:cityId, String:deviceId, SensorOutput:sensorOutput): void	Create/simulate a sensor output for a specific device in a city
createSensorOutput	(String:cityId, SensorOutput: sensorOutput): void	Create/simulate a sensor output for all devices in a city

Associations

Association Name	**Type**	**Description**
cities	HashMap<String: cityId, City>	Global storage of all of the cities under management
persons	HashMap<String: personId, Person>	Global storage of all of the persons under management
devices	HashMap<String: deviceId, Device>	Global storage of all of the devices under management

Smart City Model Service Exception

The smart city model service exception class is a multipurpose exception class used by the smart city model service object for reporting its exceptions

Properties

Property Name	**Type**	**Description**
action	String	The action being performed when the failure occurred
reason	String	The reason for the failure

Street Light *extends Device*

The street light class is a child of *Device* and implements additional parameters that are specific to what a street light might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
Brightness	Integer	The current brightness of street light

Street Sign *extends Device*

The street sign class is a child of *Device* and implements additional parameters that are specific to what a street sign might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
text	String	The current text being displayed on the street sign

Vehicle *extends Device*

The vehicle class is a child of *Device* and implements additional parameters that are specific to what a vehicle might need in the city. This class includes an overloaded constructor which takes an instance of itself and can effectively be used to create a copy of an instance of the class.

Properties

Property Name	Type	Description
type	VehicleTypeEnum	The type of vehicle the object is
activity	String	The activity currently being executed by the vehicle
capacity	Integer	The person capacity of the vehicle
fee	Integer	The fee to use the vehicle

Visitor *extends Person*

Visitor inherits from the *Person* class and represents a visitor in the smart city model service. It is not associated to a city but can be queried so that the city its currently in can be determined. The class, although an extension of *Person,* does not implement anything beyond what's required by its parent.

Enums

Name	Values
CurrentStatusEnum	READY, OFFLINE
ResidentRoleEnum	ADULT. CHILD, PUBLIC_ADMINISTRATOR
SensorEventTypeEnum	MICROPHONE, CAMERA, THERMOMETER, CO2METER
SensorOutputTypeEnum	SPEAKER
VehicleTypeEnum	CAR, BUS

Implementation Details

The core component of the Smart City Model Service is the Smart City Model Service Class. This class provides the API by which the Command Processor and other modules can manage models in the Smart City. The Smart City Model Service class should have a hash map where it stores Citys it manages and it should also include a hash map where it stores Persons it manages. Another hash map will be implemented for Devices in the smart city app although this hash map will be used only when searching for devices within a specific geographic location. The Smart City Model Service class should include methods for the creation, updating and getting of Citys, Devices and Person as well as the creation of Device's Sensor Events and Sensor Outputs.

The Device class should outline the parameters that all IoT devices must have. It's the least common denominator of all the data included in each IoT device. Each IoT device will be its own class that extends the Device class. This will include the Street Light, Street Sign, Information Kiosk, Robot, Vehicle, and Parking Space. For many of these devices, their extension of the parent Device class doesn't go beyond simply adding a few more parameters. The Device class should be an abstract class, any Device that's created must be from a class extending the Device class. This will enable the app to work well without any sort of "Device Type" parameter by instead evaluating what the class of any particular IoT device is.

The Person class will outline the parameters that all people under management by the Smart City app must have. The Person class should also be an abstract class and will then enforce a decision to be made about whether the Person is a of a Visitor or a Resident class, both of which extend the Person class. The Resident class will add a handful more parameters while Visitor class will add no more parameters or functionality than the Person class it extends. Having a Visitor class instead of simply having visitors be made from the Person class helps in determining if a Person is a Visitor as opposed to a Resident since we can simply look at the class of the person to make that determination.

The updating strategy for Devices and Persons should revolve around first getting the object by its ID. A new object of the same type is then created while passing in the old, out of date object. This effectively creates a copy of the object. The Command Processor should then find which changed fields were provided by the user and update those field on the object copy. The Command Processor then should call the update method on the Smart City Model Service which will **replace** the object in all the appropriate places.

The Location class should contain the latitude, longitude and radius of any location. Its associated with Devices, Persons and Citys. On the Location class will be the method getDistanceFromEdge() which enables getting Devices or Persons in a specific area. Once a query is made for Devices or Persons in a City's area, then the appropriate hash map (Smart City Model Service Devices map or Persons map) should be iterated through and the Location of each device is evaluated against the Location of the City using the getDistanceFromEdge() method on either of the objects Locations. If the returned value is less than 0 than that means that the objects are within the radius of each other and, since the radius of Devices and Persons are 0.0, that effectively means that the Person or Device is in the Citys radius. If the value is greater than 0 than that value tells us how far away the Person or Device is from the Citys edge.

This implementation effectively addresses all of the application requirements. With this design, through the API all of the models of the smart city app are easily manageable. Each City can easily be mutated including adding and changing IoT devices and sending simulated events and output to those devices. Persons in the smart city system are also easily manageable and can be added and updated via the API. All Devices and Persons are easily tracked using functionality in the Location class. Additionally, each Device, Person and City instance includes a blockchain account address which enables business transactions across each of the objects. This component will easily interface with the Controller, Ledger and Authentication services to allow for a seamless experience for smart city administrators.

Exception Handling

Exception handling should be done primarily in the Smart City Model Service. Much of the functionality of the other classes will be limited to working with getters and setters. The Smart City Model Service will have oversight over all Persons, Devices and City's in the system and should be able to easily identify things like colliding IDs and invalid values well before the objects are committed to any hash maps for storage. Much of the error handling in the app will focus on detecting collisions where unique IDs are required or attempts to update or get objects that don't exist. Exceptions in the Smart City Model Service will then be caught by the Command Processer where they'll be printed out into a human readable format in the console.

Testing

Testing is done using two scripts. The script provided for the assignment seemed to have to errors (some cmds were missing "account," for example), so it has been modified so that the commands provide all the parameters needed. A custom test script was also created and extends the testing of the original script to test things like error handling. This script is included and named test.script. The original script keeps the original name, "smartcity_model.script."

Risks

There are a few risks to the design of this Smart City Model Service. One risky aspect of the design is having redundant storage of devices in hash maps located in the Smart City Model Service class and in each Devices individual City. Keeping these two data structures in sync could be difficult as more functionality is added to the application and if they get out of sync it could break things like accurately querying for devices within a citys radius. Another risk is authentication. I didn't implement authentication in the Smart City Model Service yet since I want to implement something more sophisticated than a hard coded token but as it is right now, being able to manage the smart city as a non-authenticated user is a huge risk.