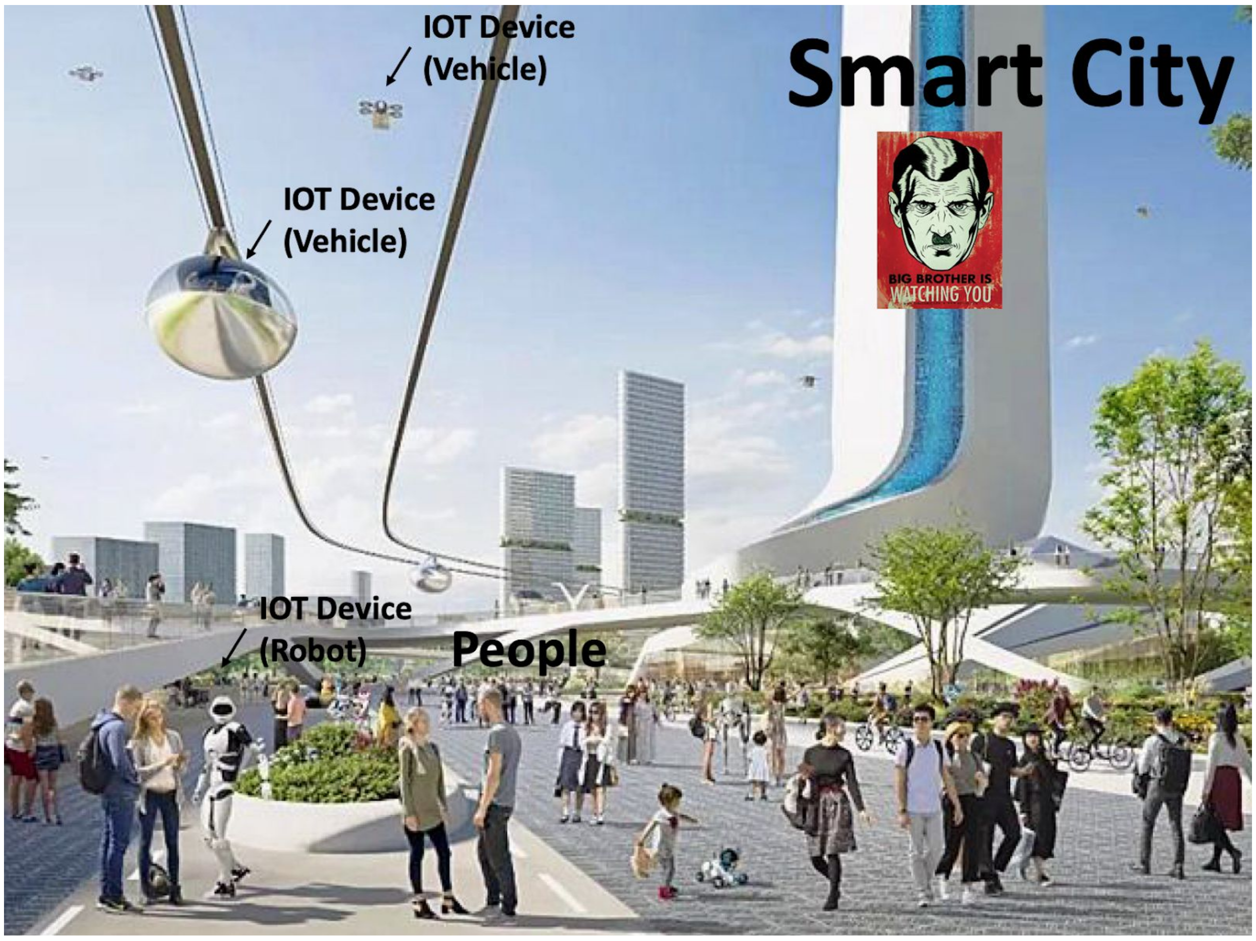
CSCI E-97 Software Design 9/15/2020

# Smart City Design Document

Date: 9/15/2020
Author: Loi Cheng
Reviewer(s): Andrew Pham, Chris Sorenson

## Introduction

The city of the future can be automated by the Smart City Software System. through AI-powered Internet of Things (IoT) devices, smart City allows city administrators to fully automate a city.  Devices include cameras, microphones, robots, and other devices. Sensors monitor the condition of people and devices. Parking spots automatically charge fees to the user's blockchain account.  Robots help residents clean the city, respond to emergencies, and help people.



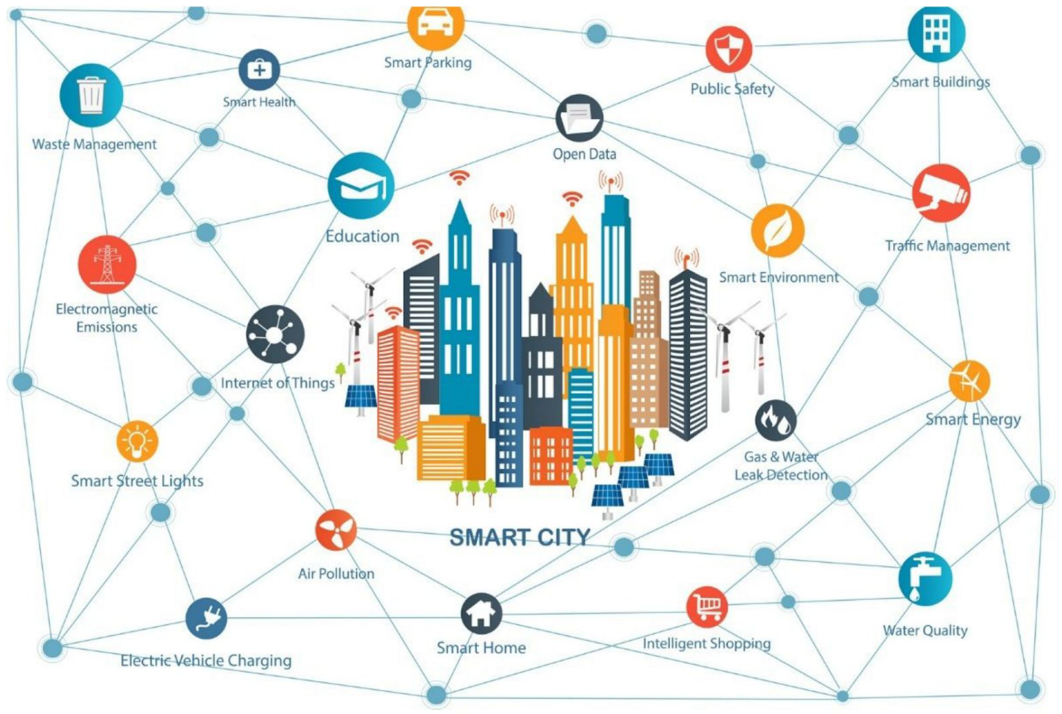
Caption: People and Devices in the Smart City

## Overview

The implementation of a Smart City Service is described. The people and devices of the Smart City are managed by the service. Devices include street signs, information kiosks, street lights, parking spaces, robots, and vehicles. The Smart City Model Service API is used to interact with those devices. The API supports getting and updating the state of the entities.

The quality of life is improved by the Smart City. These improvements include:

- Connected vehicles efficiency moves between destinations, reduces wait and transport times, improves transportation efficiency
- Smart signs and Info Kiosk brings important information to nearby people, improving direct public communications
- Smart parking spaces directs vehicles to them when empty, or directs them to other empty spaces when full
- Robots provide assistance where needed
- Sensors monitors the health of overall inhabitants and acts accordingly in emergencies



Caption: The Smart City System Interconnects the City Inhabitants with the Internet of Things

## Requirements

This section defines the requirements for the Smart City Model Service.
The Smart City Model Service is responsible for managing the state of things including:

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

### City

The City is used to model a city instance. A City has the following attributes:

- Globally unique identifier (e.g. city-1 )
- Name (e.g. "Mars Station 5")
- Multiple people, either residents or visitors
- Multiple IoT Devices
- A blockchain account for receiving and sending money
- Location (lat, long)
- Radius which specifies the area encompassed by the city, in km

### Person

Persons are 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.

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 (lat/long)

Attributes of Visitor include:

- Globally unique id
- Biometric Id
- Location (lat/long)

## Iot Devices

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

- A globally unique id
- Location(lat/long)
- Current status (ready, offline)
- Enabled (on/off)
- And the latest event emitted from the device

All IoT devices have the following sensors:

Input

- Microphone
- Camera
- Thermometer
- CO2 Meter

Output

- Speaker

The Sensors generate events that are processed by the Virtual IoT Devices. Events have a type, action, and an optional subject. For example, the microphone may generate an event {microphone,“where is the nearest parking spot?”, resident:bob}. The Camera may generate the event {camera,”person sleeping”, resident:joe}. The CO2 Sensor may generate the following event {CO2, “4000ppm”}, similarly, the Thermometer may report the current ambient temperature {thermometer, “588F”}, or the temperature of an individual {thermometer, “68.6F”, “jane”}. The speaker generates output speech, allowing the devices to interact with Residents and Guests.

IoT devices:

## Street Sign

A street sign provides information for vehicles. The city can update the text displayed on the sign. For example, it can dynamically adjust the speed limit, or warn about an alien invasion.

## Information Kiosk

The Information Kiosk helps residents and visitors. It is able to talk and display images. For example the Kiosk can display a map and help provide directions. The Kiosk can also provide small talk for bored people.

## Street Light

The Street Light can adjust its brightness through the city service. It can go from pitch black to sun replacement.

## Robot

Robots have two arms and two legs, and look like humans. Robots are mobile and can respond to commands from Residents and Visitors. For example, being an extra player in a basketball game. They can also asset in emergencies, for example, defending against an alien invasion.

## Parking Space

A Parking Space is able to detect the presence of a vehicle. A parking space has an usually outrageous hourly rate which is charged to the account associated with the vehicle. An extra second is charged as another hour.

## Vehicle

Vehicles give 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, Robots and Dogs, but requires a fee for Residents.

## Smart City Model Service

The Smart City Model Service is a top level Service interface for provisioning cities. It also controls the City's IoT devices. A command API is used to access the Smart City Model Service.

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

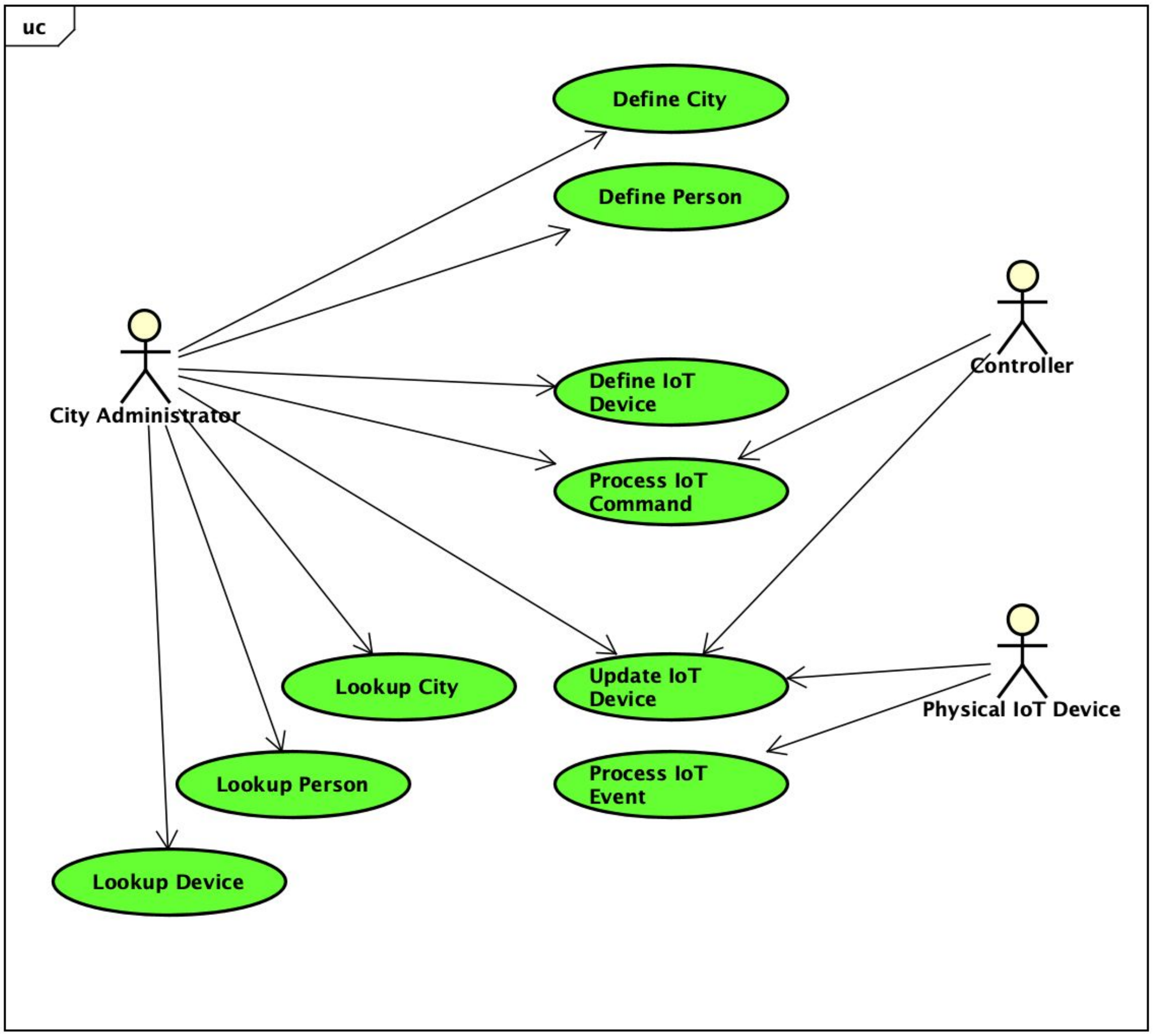
All API methods should include an auth_token parameter that will be used later to support access control.

## Command API

The Smart City Model Service supports a Command Line Interface (CLI) for configuring Cities, creating devices, 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.

## Use Cases

The following use case diagram documents the high-level use cases supported by the Smart City Software System.



Caption: UML Use case diagram with Smart City actors and use cases.

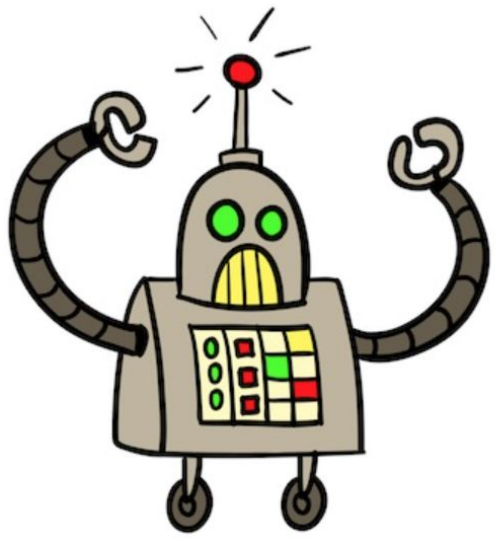
There are 5 types of actors:

- City Administrator
- Person
- Physical IoT Device
- Controller
- Simulator

**City Administrators** are humans responsible for configuring the smart city. This includes defining the city, provisioning the IoT devices, and setting up identities for the residents of the city. City Administrators should not be Robots.

**Persons** are the residents and guests that inhabit the city. Residents and visitors can interact with the various devices and request services. Residents may not vandalize or hack devices.

**Physical IoT devices** help Big Brother monitor the city. For example, one type of IoT device, the Robot Public Servant, maintains the city, including cleaning the city, arresting people, and responding to riots. IoT devices are fully automated and also respond to requests from persons and the Smart City Controller.



Caption: One type of IoT Device, the Robot

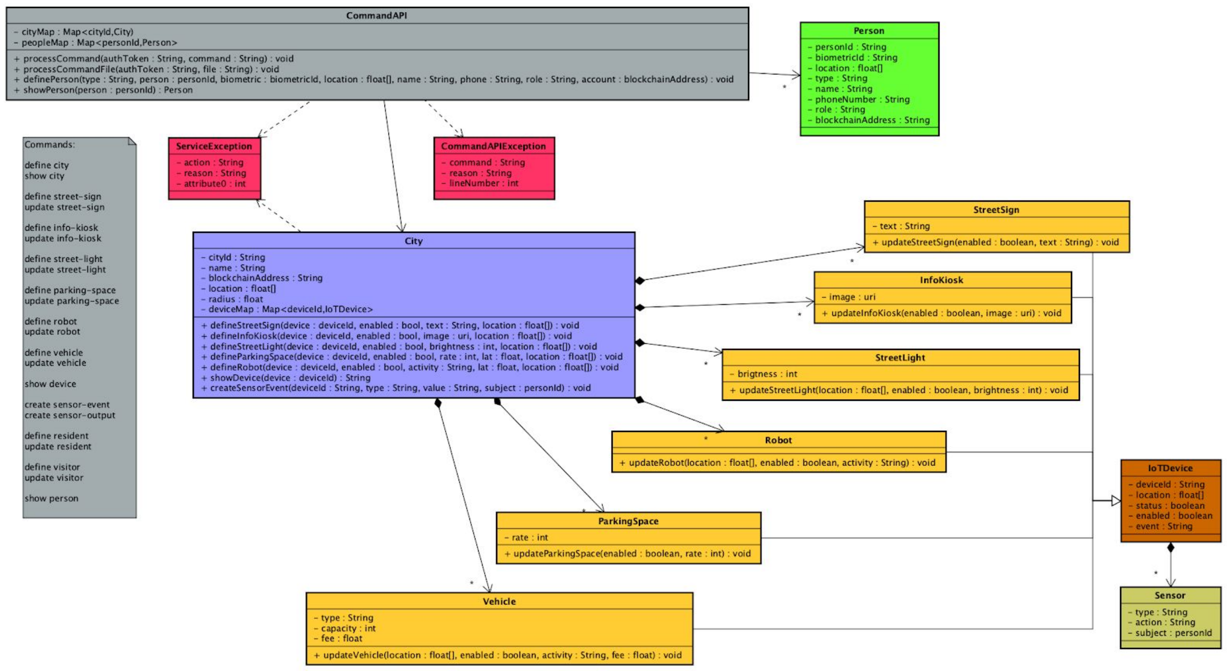
**The Smart City Controller** manages for the city. The Controller monitors the city and eavesdrops on people through IoT devices, processing crimes against the state, and generating control Commands. The Controller tracks the location and status of all persons 24/7 and keeps IoT devices near them either proactively or in response to voice commands. For example, when a person asks for a pizza, the Controller will automatically send a robot to get the pizza.

**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

### Class Diagram

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


CommandAPI
- cityMap : Map<cityId,City>
- peopleMap : Map<personId,Person>
+ processCommand(authToken : String, command : String) : void
+ processCommandFile(authToken : String, file : String) : void
+ definePerson(type : String, person : personId, biometric : biometricId, location : float[], name : String, phone : String, role : String, account : blockchainAddress) : void
+ showPerson(person : personId) : Person
Person
- personId : String
- biometricId : String
- location : float[]
- type : String
- name : String
- phoneNumber : String
- role : String
- blockchainAddress : String
Commands:
define city
show city
define street-sign
update street-sign
define info-kiosk
update info-kiosk
define street-light
update street-light
define parking-space
update parking-space
define robot
update robot
define vehicle
update vehicle
show device
create sensor-event
create sensor-output
define resident
update resident
define visitor
update visitor
show person
ServiceException
- action : String
- reason : String
- attribute0 : int
CommandAPIException
- command : String
- reason : String
- lineNumber : int
City
- cityId : String
- name : String
- blockchainAddress : String
- location : float[]
- radius : float
- deviceMap : Map<deviceId,IoTDevice>
+ defineStreetSign(device : deviceId, enabled : bool, text : String, location : float[]) : void
+ defineInfoKiosk(device : deviceId, enabled : bool, image : uri, location : float[]) : void
+ defineStreetLight(device : deviceId, enabled : bool, brightness : int, location : float[]) : void
+ defineParkingSpace(device : deviceId, enabled : bool, rate : int, lat : float, location : float[]) : void
+ defineRobot(device : deviceId, enabled : bool, activity : String, lat : float, location : float[]) : void
+ showDevice(device : deviceId) : String
+ createSensorEvent(deviceId : String, type : String, value : String, subject : personId) : void
StreetSign
- text : String
+ updateStreetSign(enabled : boolean, text : String) : void
InfoKiosk
- image : uri
+ updateInfoKiosk(enabled : boolean, image : uri) : void
StreetLight
- brigtness : int
+ updateStreetLight(location : float[], enabled : boolean, brightness : int) : void
Robot
+ updateRobot(location : float[], enabled : boolean, activity : String) : void
ParkingSpace
- rate : int
+ updateParkingSpace(enabled : boolean, rate : int) : void
Vehicle
- type : String
- capacity : int
- fee : float
+ updateVehicle(location : float[], enabled : boolean, activity : String, fee : float) : void
IoTDevice
- deviceId : String
- location : float[]
- status : boolean
- enabled : boolean
- event : String
Sensor
- type : String
- action : String
- subject : personId
*

Caption: Smart City Class Diagram

## Class Dictionary

### CommandAPI

The CommandAPI is a utility class for feeding the city a set of operations, using command syntax. This class meets the API requirements. The command syntax specification follows:

#### City Commands

# Define a city
define city <city_id> name <name> account <address> lat <float> long <float> radius <float>

# 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>

#### Device Commands

# Define a street sign
define street-sign <city_id>:<device_id> lat <float> long <float> enabled (true|false) text <text>

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

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

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

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

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

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

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

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

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

# 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>

# Update a vehicle
update vehicle <city_id>:<device_id> [lat <float> long <float>] [enabled(true|false)] [activity <string>]
[fee <int>]

# 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>]

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

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

**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>

# 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>]

# Define a new Visitor
define visitor <person_id> bio-metric <string> lat <lat> long <long>

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

# Show the details of the person
show person <person_id>
```

| **Association Name** | **Type** | **Description** |
|---|---|---|
| cityMap | Map<cityId,City> | A map of all created cities and their cityId |

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| processCommand | (authToken:string, command:string):void | Process a single command. The output of the command is formatted and displayed to stdout. Throw a CommandProcessorException on error. The authToken should be verified before processing. |
| processCommandFile | (authToken:string, commandFile:string):void | Process a set of commands provided within the given commandFile. Throw a CommandProcessorException on error. The authToken should be verified before processing. |
| definePerson | (type : String, person : personId, biometric : biometricId, location : float[], name : String, phone : String, role : String, account : blockchainAddress) : void | Define a new person. For type:visitor, these are not required: name, phone, role, blockchain address |
| showPerson | (person : personId) : void | Show the details of the person |

**IoTDevice**

The IoTDevice class represents an IoT device.  This class meets the IoT device requirements.

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

- A globally unique id
- Location(lat/long)
- Current status (ready, offline)
- Enabled (on/off)
- And the latest event emitted from the device

All IoT devices have the following input sensors:

- Microphone
- Camera
- Thermometer
- CO2 Meter

The Sensors generate events that are processed by the Virtual IoT Devices. Events have a type, action, and an optional subject. For example, the microphone may generate an event {microphone,"where is the nearest pizza?", resident:joe}. Note that the microphone is able to convert speech to text. Or the Camera may generate the event {camera,"person robbing bank", resident:bob}. Note that the microphone and camera sensors use AI to automatically identify the subject person. The CO2 Sensor may generate the following event {CO2, "400ppm"}, similarly, the Thermometer may report the current ambient temperature {thermometer, "900F"}, or the temperature of an individual {thermometer, "98.6F", "jane"}.

In addition to input sensors, all IoT devices include a speaker for generating output speech, allowing the IoT devices to control Residents and Guests using natural language.

| **Property Name** | **Type** | **Description** |
|---|---|---|
| deviceId | string | The device id |
| location | float[] | The latitude and longitude location of the device |
| status | boolean | Current status (ready, offline) |
| enabled | boolean | on/off |
| event | String | The latest event emitted from the device |

| Association Name | Type | Description |
|---|---|---|
| speaker | Sensor | The device speaker |
| microphone | Sensor | The device microphone |
| camera | Sensor | The device camera |
| thermometer | Sensor | The device thermometer |
| co2meter | Sensor | The device CO2 meter |

### StreetSign

The StreetSign is a type of an IoT device.  It extends the IoTDevice class.  This class meets the street sign text requirements.

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.

| Property Name | Type | Description |
|---|---|---|
| text | String | The text displayed on the sign |

### InfoKiosk

The InfoKiosk is an information kiosk, and is a type of an IoT device.  It extends the IoTDevice class. This class meets the information kiosk requirements.

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.

| Property Name | Type | Description |
|---|---|---|
| image | uri | A pointer to the image to be displayed |

### StreetLight

The StreetLight is a type of an IoT device.  It extends the IoTDevice class.  This class meets the street light brightness requirements.

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

| Property Name | Type | Description |
|---|---|---|
| brightness | int | The brightness of the light, from 0(min) to 100(max) |

### Robot
The Robot is a type of an IoT device.  It extends the IoTDevice class.  This class meets the robot requirements

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 nuclear meltdown.

The robot inherits all properties from the IOTDevice class.

### ParkingSpace
The ParkingSpace is a type of an IoT device.  It extends the IoTDevice class.

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.

| Property Name | Type | Description |
|---|---|---|
| rate | int | The hourly rate charged by the the parking space |

### Vehicle
The vehicle is a type of an IoT device.  It extends the IoTDevice class.  All vehicles are autonomous, and owned by the city.  This class meets the vehicle requirements

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

| Property Name | Type | Description |
|---|---|---|
| type | String | The type of vehicle (bus or car) |
| capacity | int | The maximum number of passengers |

|  |  |  |
|---|---|---|
| fee | float | The amount charged for each seat per mile |

**Sensor**
This class meets the sensor requirements

The Sensors generate events that are processed by the Virtual IoT Devices. Events have a type, action, and an optional subject. For example, the microphone may generate an event {microphone,“where is the nearest outlet?”, resident:alice}.

| **Property Name** | **Type** | **Description** |
|---|---|---|
| type | String | The type of sensor (eg microphone) |
| action | String | An action specific to the type of sensor (eg “where is the nearest bus stop?”) |
| subject | personId | An optional person the action pertains to |

**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 and can be spies, or former residents trying to thwart mass surveillance. Both Residents and Visitors are still monitored and assigned a unique person id. This class meets the person's requirements.

| **Property Name** | **Type** | **Description** |
|---|---|---|
| type | String | The type of Person (resident or visitor) |
| biometricId | String | Unique id of the person |
| location | float[] | Latitude and longitude location of person |
| name | String | For type:residient is the name of the person. For visitor it is null |

|  |  |  |
|---|---|---|
| phoneNumber | String | For type:residient is the phone number of the person. For visitor it is null |
| role | String | Role of the resident (adult, child, or public administrator). For visitor it is null |
| blockchainAddress | String | Blockchain address of the resident. For visitor it is null |

## City

The City class 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 City, must access it through the CommandAPI. This class meets the city requirements.

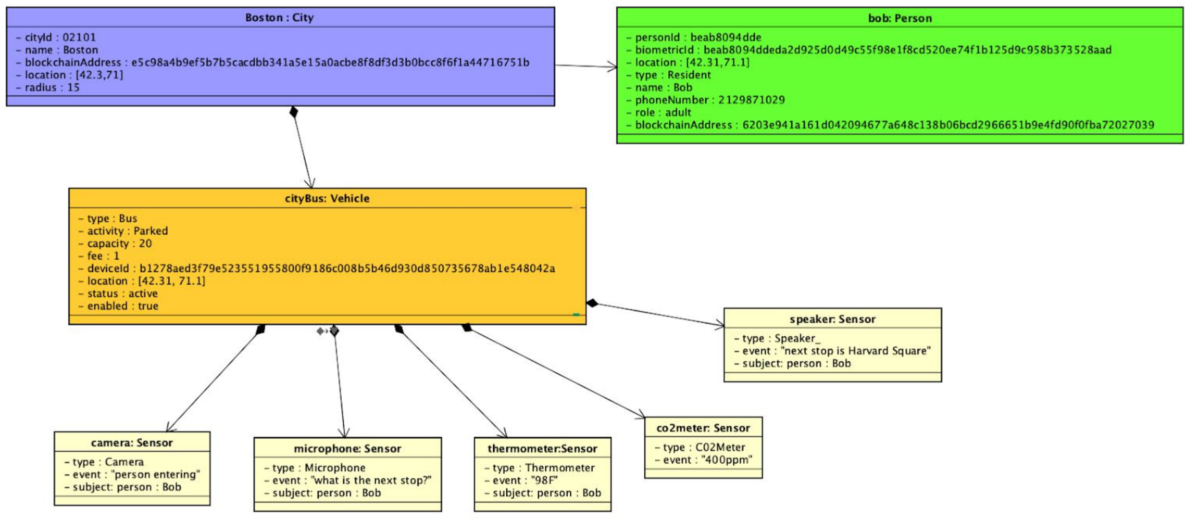
| **Property Name** | **Type** | **Description** |
|---|---|---|
| cityId | String | Unique id of the city |
| name | String | Name of the city |
| blockchainAddress | String | blockchain address of the city |
| location | float[] | Latitude and longitude location of the city |
| radius | float[] | Radius of the area encompassed by the city, in km |

| **Association Name** | **Type** | **Description** |
|---|---|---|
| peopleMap | Map<personId,Person> | A map of the person id and the associated Person |
| deviceMap | Map<deviceId,IoTDevice> | A map of the device id and the associated device |

| **Method Name** | **Signature** | **Description** |
|---|---|---|
| defineStreetSign | (device : deviceId, enabled : bool, text : String, location : float[]) : void | Define a new street sign, and add it to the device map |
| defineInfoKiosk | (device : deviceId, enabled : bool, image : uri, location : float[]) : void | Define a new info kiosk, and add it to the device map |
| defineStreetLight | (device : deviceId, enabled : bool, brightness : int, location : float[]) : void | Define a new street light, and add it to the device map |
| defineParkingSpace | (device : deviceId, enabled : bool, rate : int, lat : float, location : float[]) : void | Define a new parking space, and add it to the device map |
| defineRobot | (device : deviceId, enabled : bool, activity : String, lat : float, location : float[]) : void | Define a new robot, and add it to the device map |
| showDevice | (device : deviceId) : String | Show the details of a device, if device id is omitted, show details for all devices within the city |
| createSensorEvent | (deviceId : String, type : String, value : String, subject : personId) : void | Simulate a device sensor event. The subject is optional |

## Implementation Details

The following simplified instance diagram shows a City with 1 person and 1 device.



The core component is the City class. The City class provides an API for interacting with the City and implements the API methods that manage the people and devices.

## Exception Handling

There are 2 types of exceptions to handle: ServiceException, and CommandException.

### ServiceException

The ServiceException is returned from the methods in response to a specific error. It captures the action that was attempted and the reason for the failure.  Service exceptions should be caught by the Command Exception.

| **Property Name** | **Type** | **Description** |
|---|---|---|
| action | string | action that was performed (e.g., "update robot ") |
| reason | string | Reason for the exception (e.g. "robot not found"). |

**CommandException**

The CommandException is returned from the CommandAPI methods in response to an error condition. The CommandException 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.

| Property Name | Type | Description |
|---|---|---|
| command | string | Command that was performed (e.g., "update robot ") |
| reason | string | Reason for the exception (e.g. "robot not found"). |
| lineNumber | int | The line number of the command in the input file. |

## Testing

Implement a TestDriver class that implements a main() method. The main() method can call these methods:

CommandAPI.processCommandFile(authToken: String, file:String)
CommandAPI.processCommandFile(authToken: String)

The main method can receive a command file, or allow a user to enter commands in a console if no file is specified.   The TestDriver class should be defined within the package "cscie97.smartcity.test".

## Risks

Since this is a system dealing with people and devices, it will be subject to hackers who will attempt to undermine it. City administrators are required to have an authorization token to use the city service. The city is under mass surveillance, and the information it collects can be leaked to China if any of the devices are made in China.  So, due to national security, no devices can be made in China, and Tik Tok cannot be installed in any info kiosks.  However, the information can be freely shared with Google or Facebook to show more targeted ads to improve people's experiences.