HouseMate Model Service Design Document

Date: 9/22/18
Author: Brandon T. Wood
Reviewers: Sathish Sundar and Stephen Thompson

Introduction

This document defines the design and implementation plans for the HouseMate Service, a SmartHome manager model datastore and transfer service. This service is designed to model the state and store information about appliances, houses, rooms and occupants. That data can be edited and queried by sensors and IoT devices by making use of the HouseMate Service's API. Below are details about the implementation as it is planned along with a Use Case, Class dictionary and UML diagram to help better illustrate what behaviors and probable structure our Software's will take on.

Overview

HouseMate is a cloud based SaaS offering that manages multiple 'SmartHomes' by integrating the home's; security systems, home automation appliances, smart lights and IoT devices into a single, easy to use management console with an API and CLI. This software also allows the monitoring of electrical usage(per appliance, per room, per house) and the monitoring of multiple occupants whereabouts.

This HouseMate Model Service acts as a transfer layer and datastore for information about currently registered SmartHomes, devices and their occupants. It can take changes to the data from and allows the querying of that data through both an API and a built in CLI. The devices on the network can make use of this API to register and add data. Example: A security service using multiple cameras could integrate the rooms the cameras are in and then send updated data about what occupant is spotted in what room. This service is the core of the overall SaaS offering as it allows the integration of all the devices and a secured, single point of truth from which devices can store and send data.

Requirements

This service needs to be able to easily store and modify large volumes of data about multiple houses, occupants, rooms and smart appliances. Each of these will also need a unique user identification and the ability for users to figure out what: appliances associated which each room, rooms associates with what house, and what occupant associate with what room. The service should also be highly performant with scalability and extensibility in mind. The software underlying should be extremely simple and robust in order to keep maintainability while also allowing flexibility to accommodate for a wide range of ever expanding IoT devices.

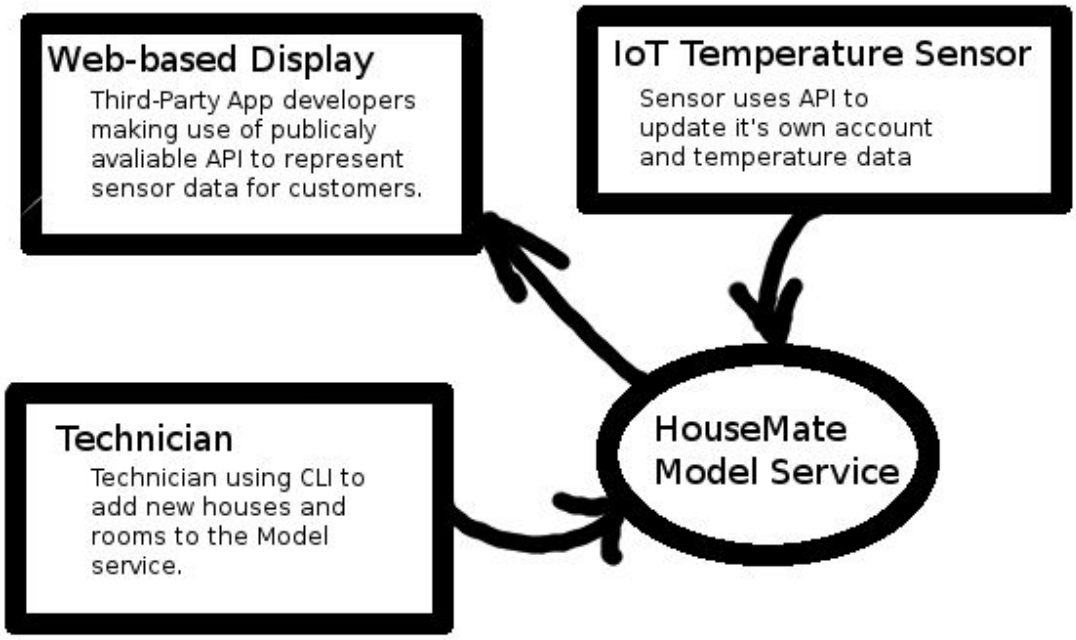
With these above things in mind I have decided to break out everything into KnowledgeGraphs, a data driven design that allows max flexibility, performance and extensibility. All data is stored in three graphs:

entityGraph - Data stored : UUID, entity type[house, room, etc], a name.
associationGraph - Data stored: a UUID, the relationship, another entities UUID.
valueGraph - Data stored: at UUID, a value type and a value.

In this way if we need to get the value of a single valueType from one room type across the entire instance we can easily cross query it from the valueGraph, which is much more efficient than having to iterate a list of nested objects. If we suddenly need a new room type, or room value over night we can backfill the value easily by iterating over the entitiesGraph and creating valueGraph entries. Lastly we could easily replace the knowledge graph with a thinner interface that stores data in a custom datastore which could greatly increase performance.

Use Cases

This design would support a wide variety of IoT devices and home configurations, for those that it does not directly support it could easily be extended to. It could be integrated with existing security systems and smart hubs to bring a holistic organization and datastore.

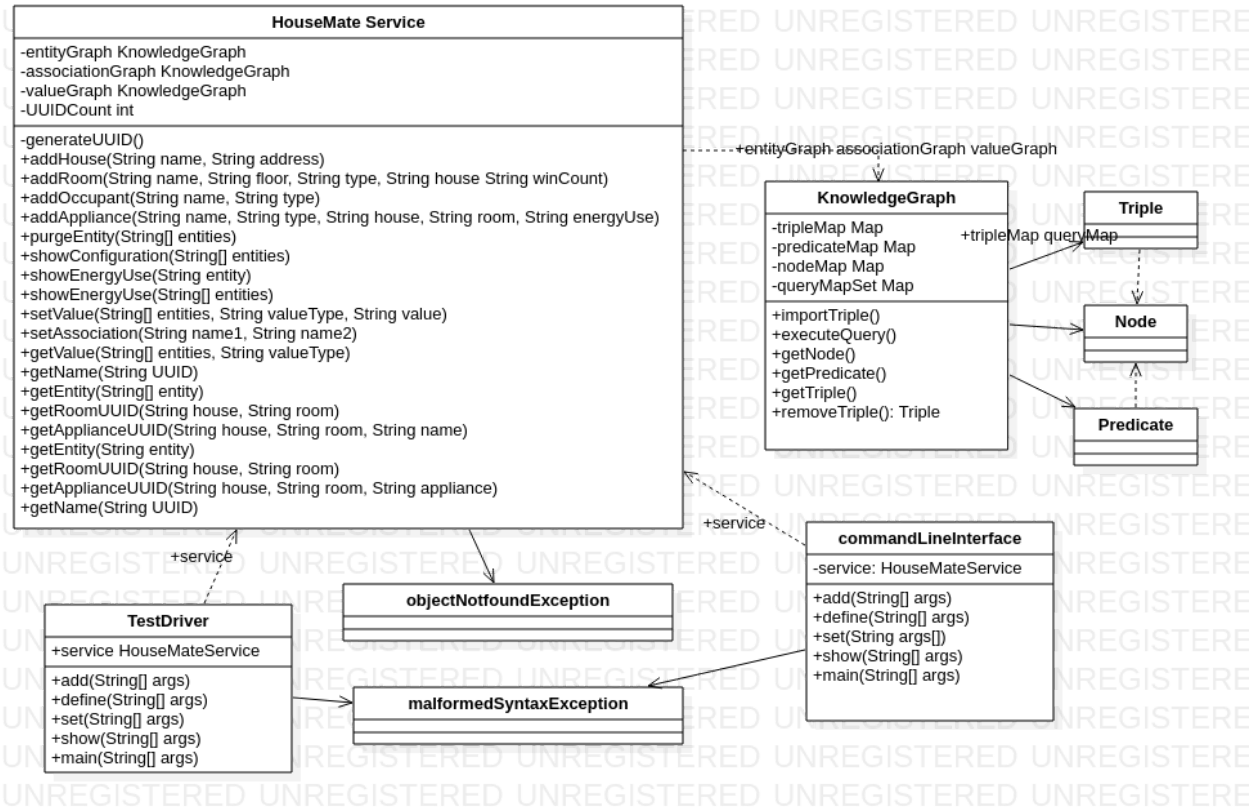


Implementation

Currently the implementation plan makes use of a modified version of an existing codebase, the Knowledge Graph, which is a queryable datastore where sets of three values are paired together and it has a custom and simple query language. In this case we will have three KnowledgeGraphs, an **entityGraph** which stores all of the rooms, houses, appliances and occupants as a entity-type:name:UUID triples, an **associationGraph** that holds triples for all relations between entities such as room5(UUID):inside:houseB(UUID) and a **valueGraph** that stores all values related to the entities as a EntityUUID:ValueType:Value triples.

Calls through the CLI will be parsed and used to call API methods that will populate or query the above graphs. There will also be methods that instantiate each of the different entity types named: addRoom, addHouse, etc. And there will be a function to create and edit values in the valueGraph and also purge out entities from the graphs.

Class Diagram



Class Dictionary

KnowledgeGraph

For further details on the knowledge graph please see it's related design document. The only change should be that query syntax checks were internalized so as to simplify the modularity of the object and that we will not be treating it like a singleton.

HouseMate Service

The main service class, initiates all the graphs, exposes the REST API and orchestrates all the incoming and outgoing data. Also has a function to generate unique user IDs or UUIDs that are unique within the service.

Method Name	**Signature**	**Description**
generateUUID	Int	Returns the next UUID and then increments

		the counter.
addHouse	void	Adds a new house entity to the entityGraph and maps corresponding values to the valueGraph. Takes a name and address as Strings.
addRoom	void	Adds a new room entity to the entityGraph and maps corresponding values to the valueGraph. Takes a name, floor, type, house and window number as Strings.
addOccupant	void	Adds a new occupant entity to the entityGraph and maps corresponding values to the valueGraph. Takes a name and type as a String.
addAppliance	void	Adds a new appliance entity to the entityGraph and maps corresponding values to the valueGraph. Takes a name, type, house, room and energyUse as Strings.
purgeEntity	void	Removes all values, associations and entities related to a passed entity name; house, room, etc.
setValue	void	Creates a new entry in valueGraph mapped to an entity by it's UUID. Takes a name, valueType and value.
showConfiguration	String	Returns a given entity, it's values, any of its sub entities and their values. Can be used to print out all values or return only certain entities.
showEnergyUse	String	Returns in watts the summation of a given entity's energy use and its sub-entity's energy use. Takes either nothing for total power or an entity name for it and it's summated sub entities energyUse.
setAssociation	void	Creates an entry in the associationGraph between the two name Strings passed to it.
getValue	String	When passed an entity and a value type will return the value for that pair.

Property Name	Type	Description
entityGraph	KnowledgeGraph	Stores a dataset of all entities known to the service as UUID:name:entitytype triples.
associationGraph	KnowledgeGraph	Stores a dataset of the relations between all entities as UUID1:association:UUID2 triples.
valueGraph	KnowledgeGraph	Stores a dataset of all values for entities as UUID:valuetype:value triples.
UUIDCount	Int	Stores the current value of the next UUID to be assigned to an entity to make them unique.

Exception Handling

Exception handling is focused on syntax and parameter count in the commandLineInterface class, when the wrong parameter count is given it throws a malformedSyntax exception. Additionally exceptions are thrown when trying to relate non existent entities which causes a objectNotFoundException to be thrown.

Testing

Included with the testDriver which takes a file of line by line CLI commands and runs them against the interface, I made it so the driver also prints out what line it is working on before it completes the work for readability. There is the TestScriptDriver which covers some of the instances illustrated in the requirements test script, there is also a crude proof of concept CLI in the commandLineInterfaces class that can be tested when it's main is run. Lastly there is an additional FunctionalTestDriver created during development that further tests the software and proves exception handling.

Risks

Because we are using an in-memory data store implementation with no writing to disk all changes to the datastore reset on service shutdown. In the future we could use a fully loaded database like Postgresql or a datastore like Redis. Additionally fuzzing and API injections were not practiced meaning there could be many vulnerabilities.