Communicating with the Nest thermostat with OpenDOF

Recently I wrote a simple java app to bridge communication between the Nest thermostat and OpenDOF. This required writing an interface, a provider and requestor, and a secure proxy server. It also required learning about the Nest API as well as the communication protocol used to interface with the Nest thermostat, in this case Firebase.

OpenDOF Interface

When writing an interface that represents some real-world object, with data and behaviors, it's important to keep it as small as possible without restricting any of the actions and data you want to provide. The Nest API details a few dozen fields that the thermostat has available, including various temperatures, both in Fahrenheit and Celsius, connection status values, various ids, etc. The brute-force approach would have been to translate this API directly to an interface an OpenDOF provider could use. However, in my case, this would have been a mistake. There are a lot of properties that wouldn't fit nicely into our data model or we just might not care about (e.g. both a "name" and a "long name").

My recommendation is to create an interface based on the minimum amount of data and behaviors you *expect* to use, but be flexible and be ready to add or delete properties should the need arise during development. Writing a minimum amount of properties also allows for easier future re-use.

When just starting out, write only the properties. Various methods, events and exceptions *could* be written beforehand based on expected behaviors, but it was most helpful to add these during development of other parts of the application. The reason is three-fold:

First, it helps with incremental development. With just the minimum of necessary information it's easier to get a prototype up and running quickly.

Second, there may be unexpected behaviors that necessitate a new method, event or exception. For example, after some development on the Nest protocol bridge I realized that Nest would not allow a target temperature to be set if the Hvac mode was set to either "off" or "heat-cool". The app would seemingly set the target temperature, but it would immediately be set back to its original value. This was not documented in Nest's API, and there were more instances like this. So I went back to my Nest interface and added exceptions to handle these behaviors.

Lastly, there were events I originally thought we would have to provide, such as handling the event of the ambient temperature dropping below the target temperature (meaning the device should turn the heat on). So I wrote this event into the interface. However, this was completely unnecessary as the Nest thermostat itself handles this event. Eventually, I had to go back to the interface and delete this portion.

Relying on External Libraries

The Nest API uses Firebase client libraries to provide real time data values from the Nest thermostat to external services, such as our OpenDOF provider protocol bridge. So it is necessary to know some amount of information about this library. After writing this application, I believe it would have been advantageous to gather more information about the Firebase library before I even began implementing the OpenDOF provider or requestor. It turns out that in this case there is a basic tutorial and reference guide available for Firebase that I could have utilized in more depth that would have made development easier. After writing a significant portion of code, I discovered that with too many requests for data through Firebase, their protocol throttled by provider and a timeout of approximately 5 minutes would need to expire before I could receive more data. This wasn't clear during testing, however, and knowing more about Firebase beforehand would have made debugging easier.

Some problems can arise from this reliance even with thorough research, however. For example, there seemed to be a critical bug in the latest release of the Firebase library that would repeatedly try to set the target temperature in an infinite loop. Eventually, I had to use a prior version of the library in order to get the correct functionality. Be aware that for every external component required there could be bugs / oddities that might be difficult to locate, and be ready to experiment with different versions of the library or different libraries in particular. In this case, I had several different options, such as parsing JSON objects received from HttpRequests.

Keep Optimizations in Mind

In an earlier version of the application, the OpenDOF provider didn't store any data values locally. Any time the requestor asked for information, the provider requested it from the Nest thermostat through Firebase. This wasn't a problem until many more properties were added, and suddenly the Firebase protocol would throttle my application to reduce network requests. Furthermore, many of these properties just didn't make sense to repeatedly get from the thermostat itself. For example, the thermostat name, device ID, software version, etc. could be requested just once, at the beginning of the connection. A better solution was to keep values stored locally on the provider, and for those properties that are not expected to change, retrieve them locally.

Additionally, my original design had a request for a DataSnapshot (stored as a JSON object) for every data value whenever any value changed in real time. This was inefficient and created too many requests, which led to throttling issues (discussed above). A better solution was to receive this DataSnapshot just once whenever a value changed in real time, then iterate through it to update all the necessary data values. By keeping processor and network efficiency in mind I could have avoided this problem entirely.

Security

The Nest API uses OAuth 2.0 for authorizing a client (our protocol bridge application) to communicate with their servers as part of their security protocol. This means the communication between Firebase and the Nest servers is secure; it does not mean that any communication between our provider and requestor(s) is secure; this still needs to be handled through OpenDOF's Authentication Server protocol. The point is that you should not assume *all* security concerns are handled for you should their protocol provide measures for network security.

Modularity

Keep modularity in mind when developing a protocol bridge and try to keep distinct behaviors separate. By adhering to the principal of Separation of Concerns, my implementation of the protocol bridge to Nest allows for easy replacement of Nest devices. For example, with very slight modification to the source code the app can easily interface with Nest thermostats, smoke alarms, and CO alarms. Additionally, it would be trivial to replace the thermostat used in this app for any device with data values that can be accessed via JSON objects.

Reference Section

Some web sites I found helpful during development of the Nest thermostat protocol bridge:

https://developer.nest.com/documentation/api-reference

https://www.firebase.com/docs/java-api/javadoc/index.html

https://www.firebase.com/docs/web/guide/

http://en.wikipedia.org/wiki/Representational_state_transfer