Crownstone-Homey Integration

Mart-Jan Koedam

September 2020

1 Cloud

This chapter will contain all the information about the development of the custom Crownstone app for Homey using the crownstone-cloud.

1.1 App Manifest

Before creating an application in Homey, it is important to know how a Homey app works. A good way to see that is to look at the file structure of the app. If the file structure of the app is easy to understand, it is much easier to find what you need. On the Homey Developer website tutorial, an example of the file structure is shown:

```
com.athom.example
├── README.txt
├── api.js
├── app.js
├── app.json
├── assets
│   ├── icon.svg
│   └── images
│       ├── large.png
│       └── small.png
├── drivers
│   └── my_driver
│       ├── assets
│       │   ├── icon.svg
│       │   └── images
│       │       ├── large.png
│       │       └── small.png
│       ├── device.js
│       └── driver.js
├── env.json
└── locales
    └── en.json
```

```
│
│   └── nl.json
└── settings
    └── index.html
```

Figure 1: A template of the file structure of a Homey app [1].

The most important parts of the file are `app.js`, which is the starting point for the app, `app.json`, which contains all metadata for the app, `drivers`, which contains all the data for the drivers (in this case for Crownstone) and `settings`, which makes it possible for users to use their credentials to make a connection to the cloud. All of these files will be explained in more detail later on.

1.2 App

1.2.1 Initiation

The app.js file is the starting point of the application. As soon as the Homey starts, the *OnInit()* method is called. The first thing that will happen is getting the credentials (only the username and password are needed) for the Crownstone Cloud from the user which can be obtained from Homey.ManagerSettings, where the credentials are saved. This was done in the settings which will be explained in more detail in the section *App Settings*.
The credentials can be obtained using the method *ManagerSettings.get()* and can be saved in a variable like this:

```
this.email = Homey.ManagerSettings.get('email');
this.password = Homey.ManagerSettings.get('password');
```

1.2.2 Setting up the connections

The next step is to make all the necessary connections. This is done using the *setupConnections(email, password)*-function. The arguments of the function are the email-address and the password for the user's Crownstone account. This function calls multiple functions which are explained below.

`Crownstone Cloud`
First, a connection to the Crownstone Cloud will be made using the Crownstone Cloud library:

```
const cloudLib = require('crownstone-cloud');
const cloud = new cloudLib.CrownstoneCloud();
```

Then, this method is used to make a connection to the Crownstone Cloud:

```
cloud.login(email, password)
```

`getPresentPeople`
After that, all the users and their current locations in the sphere should be obtained from the cloud. The data will be used later for the Flows.

The first thing the *getPresentPeople()*-function will do is obtain the current sphere ID using the *getSphereId()*-function.
If the user is not present in the sphere or has his location turned off, the sphere ID will be `undefined` and the list of user locations will stay empty.
If the sphere ID is defined, the data will be obtained from the Crownstone Cloud and saved in the local variable *userLocations* using:

```
await cloud.sphere(sphereId).presentPeople();
```

`Event Server`
The Event Server is used to receive events like 'a user who enters or leaves a certain room'. These events are used for the Flows.
The Crownstone SSE Library is used to make a connection with the event server:

```
const sseLib = require('crownstone-sse');
const sse = new sseLib.CrownstoneSSE();
```

To log in to the event server, the function *loginToEventServer(email, password)* is used with the email-address and the password for the user's Crownstone account as parameters.
First, all possible running eventHandlers get stopped, in case a new connection has been made using *sse.stop();*.
Then a new connection with the event server is made using the email-address and password using the method *sse.login(email, password);*.
And then the eventHandler will be started using *sse.start(eventHandler);*. More information on how the eventHandler is used will be provided in the section of Flows.

`Obtain user locations`
After that the function *obtainUserLocations()* is called, this function will call the *getPresentPeople()*-function every 30 minutes in case of missed events.

1.2.3 Eventlistener

Every time a user changes the credentials in the settings, a new connection should be established with the cloud in case the user entered the wrong credentials before or changed his password for the cloud. In order to notice that the credentials have been changed, an event listener is added which will fire when a setting - such as a username or a password - has been 'set':

```
Homey.ManagerSettings.on('set', function(){}
```

When it fires, the newly-stored credentials will be obtained from the Homey and the *setupConnections(email, password)*-function is called which will establish a new connection to the cloud. This process is described in the sequence diagram below:

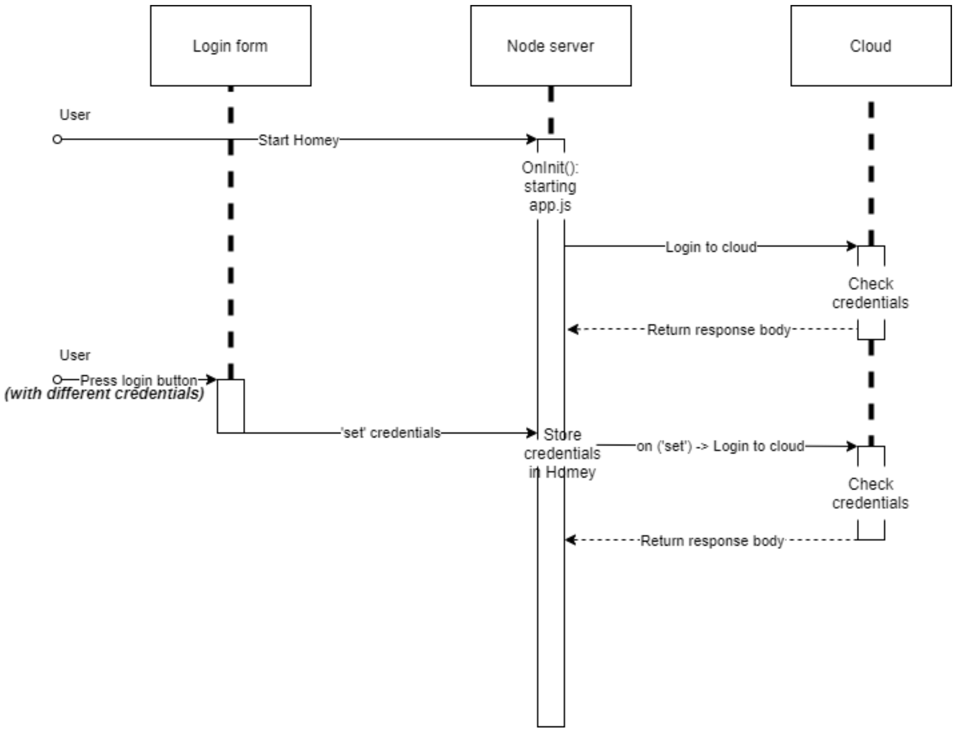

Figure 2: Sequence Diagram of the app making a connection with the cloud

1.3 Driver

When making the Homey App for Crownstone, a driver is needed to add support for devices like the Crownstone plug. The driver manages the addition and working of the devices. There is only one Driver instance and as many Device instances as there are devices added to the Homey.

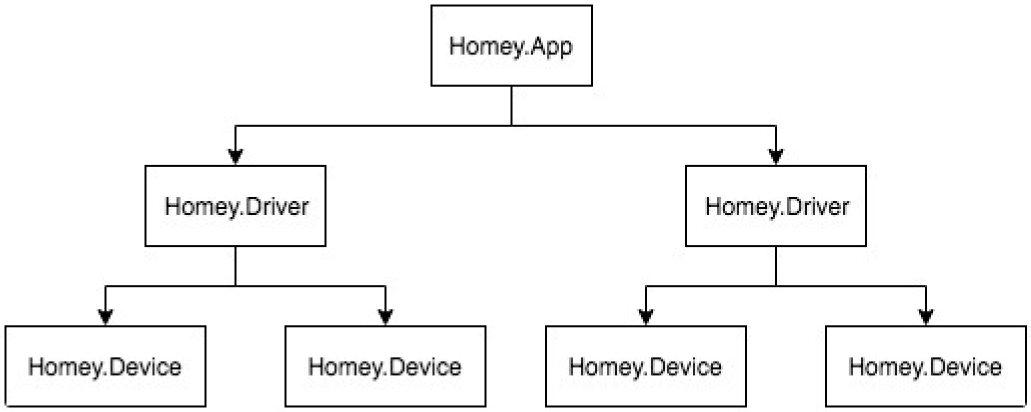

Figure 3: The structure of the drivers and devices [2].

1.3.1 Driver

Adding devices

When a user is adding a device and the 'list_devices' view is called, the method *onPairListDevices(data, callback)* is called. This method will call the *getLocation()*-function which is defined in the app. That function will return the cloud-instance and the sphere ID. Then the *getDevices(cloud, sphereId)*-function is called which uses the instance of the cloud and the sphere ID to obtain all the devices in the current sphere using:

```
await cloud.sphere(sphereId).crownstones();
```

Then, the function *listDevices(deviceList)* is called which will create empty *device* objects and adding the properties of the devices obtain with the cloud to it:

```
let device = {
    'name': crownstoneList[i].name,
    'data': {
        'id': crownstoneList[i].id,
    }
}
```

After that a list in this format of all the devices in the sphere is returned and will be shown to the user. It is important to keep in mind that the properties added to the object must be static, because the Homey identifies different devices using the object, instead of the id only.

1.3.2 Device

Initiation

When the device is initialized, it will obtain the cloud instance from the CrownstoneApp using:

```
this.cloud = Homey.app.getCloud();
```

After that, a capability listener will be registered to listen for device state changes:

```
this.registerCapabilityListener('onoff', this.onCapabilityOnoff.bind(this));
```

switch state

If a state change is requested for a device a request will be sent to the cloud.
If the device is requested to turn on:

```
await this.cloud.crownstone(this.getData().id).turnOn()
```

And if the device requested to turn off:

```
await this.cloud.crownstone(this.getData().id).turnOff()
```

1.4 Internationalization

Internationalization doesn't look that important in the beginning process of development, but later on, it's getting more important, especially if the application will be accessible for users around the world. That's why the implementation of internationalization early on in the development is important since it only gets more difficult to implement internationalization later on in the development of an application.
All the different translations that are supported in the app will be included in the `/locales/` folder (see Figure 1). The translation strings are stored in a .json-file with the language code as their name. For example: `nl.json` and `en.json`.

2 Flows

User can create Flows for their Homey in the Flow Editor using different cards.
There are three types of cards: when, and, then.
The when.. cards are called triggers.
The ..and.. cards are called conditions.
The ..then cards are called actions.
For the Crownstone App, only the trigger and condition card are used.

2.1 Trigger card

The trigger card will fire when a new event has occurred. In this case, when a specific user enters a specific room, which will run all of the user's flows that use this trigger.
The trigger is: *"When [user] enters [room]"*, with *[user]* and *[room]* the variables selected by the user.

2.2 Condition card

The condition must be true, for the flow to continue. In this case, a check will be done to see if a specific user is or isn't in a specific room.
The condition is: *"When [user] {is|isn't} present in [room]"*, with *[user]* and *[room]* the variables selected by the user and *{is}* the normal state and *{isn't}* the inverted state.
If the condition state is inverted, the condition must be false, for the flow to continue.

2.3 Users and Rooms

For both the trigger- and condition-card, the user can choose a specific user or room in the sphere for the card. When a user selects the specific user or room, an array with options appear. This is done by using the autocomplete-argument. The array with options is obtained using the code in the app.

`Users`
To obtain the users in the current sphere, the *getUsers()*-function is called. This function first obtains the sphere ID of the current sphere and will then obtain a list with all the users and their information in the sphere using:

```
const users = await cloud.sphere(sphereId).users();
```

Then, the function *listUsers(users)* which will return a list with only the username and ID is called. The first thing this function does is create a default user-object named *'Somebody'* with *'default'* as ID. This value can be used if a trigger or conditions applies for *every* user in the room. For example: When *somebody* enters a room..
After that the function will call the *addUserToList(userList, users)*-function

three times, since the original user list is divided in three sections: admins, members and guests. The argument *userlist* is the list that will be returned from this function and the argument *users* is the list with admins, members or guests. The *addUserToList(userList, users)*-function will create a new user-object and will set the user's first- and last name as the name and set the user's ID to be the ID like this:

```
const user = {
    name: users[i].firstName + ' ' + users[i].lastName,
    id: users[i].id,
};
```

This will be pushed to the list that will be returned by the *listUsers(users)*-function.
If no users are found, the length of the list will be one, since the default user *'Somebody'* is always defined. So if the length of the user list is not greater than one, an empty list will be returned.

`Rooms`
The way to obtain the rooms is almost the same as that of the users. First, the *getRooms()*-function will be called, which will obtain the sphere ID of the current sphere and will then obtain a list with all the rooms and their information in the sphere using:

```
const rooms = await cloud.sphere(sphereId).locations();
```

When the rooms are obtained, the function *listRooms(rooms)* with the list of rooms as parameter will be called to create a list of the rooms that will be shown to the user. A new room-object will be created for every room and the room's name will be set as the name, and it's ID will be set as the ID like this:

```
const room = {
    name: rooms[i].name,
    id: rooms[i].id,
};
```

If there are no rooms found, an empty list will be returned.

2.4 Event-Handler

The Event-Handler will receive events when a user enters or leaves a room. When an event has been received, the function *runTrigger(data, entersRoom)* will be called, with the parameter *data* as the data received in the event and the parameter *entersRoom* as a boolean which is true if a user enters a room, and is false if a user leaves a room.

2.4.1 Update userlocation list

The users and their location is saved in the local variable *userLocations* which will prevent the need for having to query the users and their locations from the

cloud every time an event occurs. This will give the triggers a faster response time and will also make it easier to debug.
For every new event, the local variable needs to be updated which is done by using the function *updateUserLocationList(entersRoom, userId, location)* with entersRoom as a boolean that determines if a user enters or leaves a room, the userId as the user ID, and the location as the location the user just entered or left.
First, the function *checkUserId(userId)* will check if the user is already defined in the list. If it is, it will return the index of the list where it is in, and if it isn't, it will return -1. This value is saved in the variable *userInList*.
After that, if *entersRoom* is true, and the user just entered a room, there will be checked if the user is already in the list or not. If the user is not yet in the list, a object of the user with it's location will be added to the list. If the user is already in the list, there will first be a check if the location in the list is the same as the location of the user. In that case, an event probably has been missed, and the *getPresentPeople()*-function will be called to reset the variable.
If the location in the list is not the same as the new location, that location will be updated with the new location.
If *entersRoom* is false, and the user just left a room, there will be a check if the user is in the list. If the user is in the list and the location is the same as the location the user just left, an event probably has been missed, and the *getPresentPeople()*-function will be called to reset the variable. Otherwise, the list is already up-to-date and the function will end.

2.4.2 Triggers

After the list has been updated in the *runTrigger()*-function, and the event is a user *entering* a room, the *presenceTrigger* will be fired with the current state as argument. The state is an object which contains the user ID and the location ID.

`Trigger`
For the Trigger in the *registerRunListener*, the flow will run if the location ID and user ID in the state are equal to the room ID and the user ID in the arguments of the Trigger-card, or the location ID in the state and the room ID in the argument is equal and the user ID is equal to *'default'*, the return value is true and the flow will run:

```
Promise.resolve(args.rooms.id === state.locationId
     && args.users.id === state.userId
      || args.users.id === 'default')
```

`Condition`
If the return value of the Trigger was true, and a condition-card is defined, a check for the condition in the *registerRunListener* will be made.
First, the code checks if the user ID argument is *'default'*, if that is the case,

the function *checkRoomId(roomId)* will be called to check if the room ID exists in the list of userLocations so that if *'Somebody'* is in that specific room, the flow will run. The function returns true if the room exists in the list and false if it doesn't.
If the user ID is specified, the function *checkUserId(userId)* will be called to check if the user is already defined in the list. If the user doesn't exist in the list, the code will return false, but if the user exists in the list, it will check if the room ID argument is equal to the current location of that user which will return true or false:

```
return Promise.resolve(args.rooms.id
    === userLocations[userInList].locations[0]);
```

References

[1] Homey. App manifest.

[2] Homey. Drivers.