# SMIoT - Android Framework Guidelines

Ilse Bohé, Michiel Willocx, Vincent Naessens

imec-DistriNet
KU Leuven, Technology Campus Ghent
Gebroeders de Smetstraat 1, 9000 Ghent, Belgium
firstname.lastname@kuleuven.be

## 1 Architectural Design & Use of the Framework

A short recap of the important architectural blocks and how to use them in the Android implementation of the SMIoT Framework (`https://github.com/ku-leuven-msec/SMIoT-Framework`). More information on the design can be found in the report *SMIoT: A Software Architecture for Maintainable Internet-of-Things Applications*

### 1.1 Virtual IoT Device Layer

**Virtual IoT connector** The Virtual IoT Connector takes care of the communication with the physical device. It is thus dependent on the technology used by the vendor. Note that, depending on the type, one connector can communicate with multiple devices. (e.g. one Philips Hue bridge, with multiple lamps). An example for adding a new Philips Hue connector is as follows. The first argument is always the unique systemID, other arguments depend on the type of connector.

```
VirtualIoTConnector vc = new HueGateway("HGW1", Map.of("ip",
    [redacted], "authID",
    [redacted]));
```

**Virtual IoT Devices** Each physical device in the infrastructure is represented by a Virtual IoT Device in the framework. These Virtual IoT Devices use Virtual IoT Connectors to communicate with the physical devices. Adding a new IoT Device can be done using the code below, hence the connector used to communicate with the physical device must be instantiated first. Note that due to the underlying differences between the technologies, the instantiation of the Virtual IoT Devices contains device specific code. To see which settings must be used, it is best to take a look at the examples in the code. The Virtual

IoT device can now be used to communicate with the physical device, through the provided connector.

```
Lamp l = new HueLamp("LAMP1", <vendor-specific-settings>, vc);
```

## 1.2 Component Layer

Components represent assets in the domain (e.g. a room and patient in a care environment). These components are highly dependent on the application domain, hence a room in one application might not have the same type of devices attached as a room in another application. Creating components in the framework is done using Java's standard Object Oriented Programming concepts. To increase portability of the components the only requirement is that they extend the provided `Component` class. Note that components in the framework are completely independent of the used technologies in the infrastructure. An example for a `Room` component with one `Lamp` can be found below.

```
public class Room extends Component {

    Lamp lamp;

    public Room(String componentname) {
        super(componentname);
    }

    public void setLamp(Lamp lamp) {
        this.lamp = lamp;
    }

    public void lightOn(OnRequestCompleted<Boolean> orc) {
        lamp.turnOn(orc);
    }

    public void lightsOff(OnRequestCompleted<Boolean> orc) {
        lamp.turnOff(orc);
    }

    public void changeBrightness(int brightness, OnRequestCompleted
         orc) {
        lamp.changeBrightness(brightness, orc);
    }


    public void changeColor(String color, OnRequestCompleted orc) {
        lamp.changeColor(color, orc);
    }
}
```

## 1.3 Application Environment

The application environment contains the structure of the IoT Environment. Hence, it is the point of contact for the Android Application to communicate with the IoT Environment. The configurations of the environment can be maintained on a server, but for demonstration purposes the Java class `ApplicationEnvironment` is provided. Objects of this class are able to retrieve data from a structured `JSON`-file. Several methods are provided to load the entire

environment (i.e. creating the components, connectors and devices). All components, and devices are then available via the instantiated `ApplicationEnvironment` object.

**JSON file structure** To use the `ApplicationEnvironment` class to load the IoT environment, the `JSON` must be structured as follows.

```
{
  "connectors": [
    {
      "systemID": <unique-id>,
      "type": <connector-type>,
      "settings": {
        <connector-dependent-settings>
      }
    }
  ],
  "devices": [
    {
      "type": <device-type>,
      "model": <device-model>,
      "systemID": <unique-id>,
      "settings": {
        <connector-dependent-settings>
      },
      "connector": <system-id-of-connector>
    },
    ...
  ],
  "components": [
    {
      "componentName": <unique-name>,
      "childComponents": [
        {
            "componentName": <unique-name>,
            "childComponents": [...]
            "devices": [ ]
        }
      ],
      "devices": [<unique-id-of-device>, <unique-id-of-device>,
          ...]
    },
    ...
  ]
}
```

The `<unique-id>`, is a unique identifier in the system that can be chosen by the developer. The `<connector-type>`, `<device-type>` and `<device-model>`, are predefined constants used in the framework to identify the connector type, device type and device model (dependent on vendor), respectively. The constants defined in the framework can be found under `systemmanagement/constants/Connector_constants`, `.../Device_constants` and `.../Model_constants`.

A full example of a configuration file for one room (`Room`) and 1 Philips Hue Lamp (`Lamp1`), using a Philips Hue Connector (`HGW1`) can be found below.

```
{
  "connectors": [
   {
      "systemID": "HGW1",
      "type": "HUE_CONNECTOR",
```

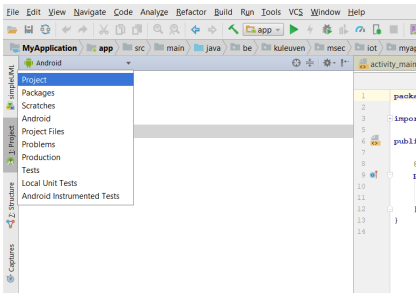
```
      "settings": {
        "ip": [redacted],
        "authID": [redacted]
      }
  ],
  "components": [
    {
      "componentName": "Room",
      "childComponents": [
      ],
      "devices": [
        {
          "type": "lamp",
          "model": "HUE",
          "systemID": "LAMP1",
          "settings": {
                "uniqueID": [redacted]
          },
          "connector": "HGW1"
        }
      ]
    }
  ]
}
```

`ApplicationEnvironment`

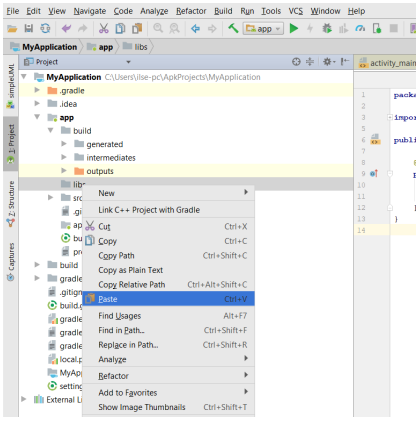
```
ApplicationEnvironment e = new ApplicationEnvironment(this);

e.getConfigurationFromServer(this, "configurations.json", new
    OnRequestCompleted<Boolean>() {
    @Override
    public void onSuccess(Boolean response) {
        e.loadEnvironment(new OnRequestCompleted<Boolean>() {
            @Override
            public void onSuccess(Boolean response) {
                [...]
            }
        });
    }
});
```

# 2 Adding the SMIoT Framework Library (`iotframework.jar`) to Android

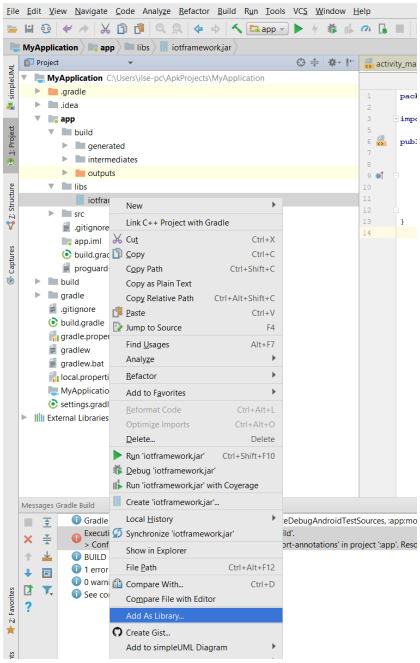
1. Change view to `project`.
2. Paste the library in the folder `app/buid/libs`
3. Add the file as library. Right click on the jar-file, click on '`Add As Library..`'. This will automatically add the library in `build.gradle`. The library can now be used by the project.




(a) Project View




(b) Paste in `libs` Folder




(c) Add As Library