Smart Home SDK

Development Guide

Version 1.0

October 31, 2014

Samsung Smart Home Partnership Program Proprietary

Table of Contents

1. Installation

1.1. Prerequisite

As of now, Smart Home SDK only supports Linux 32 bit development environment and Plug-in runs on Eclipse platform.

1.2. JRE Installation

For Eclipse to work properly, it is mandatory to have installation of JRE (Java Runtime Environment).

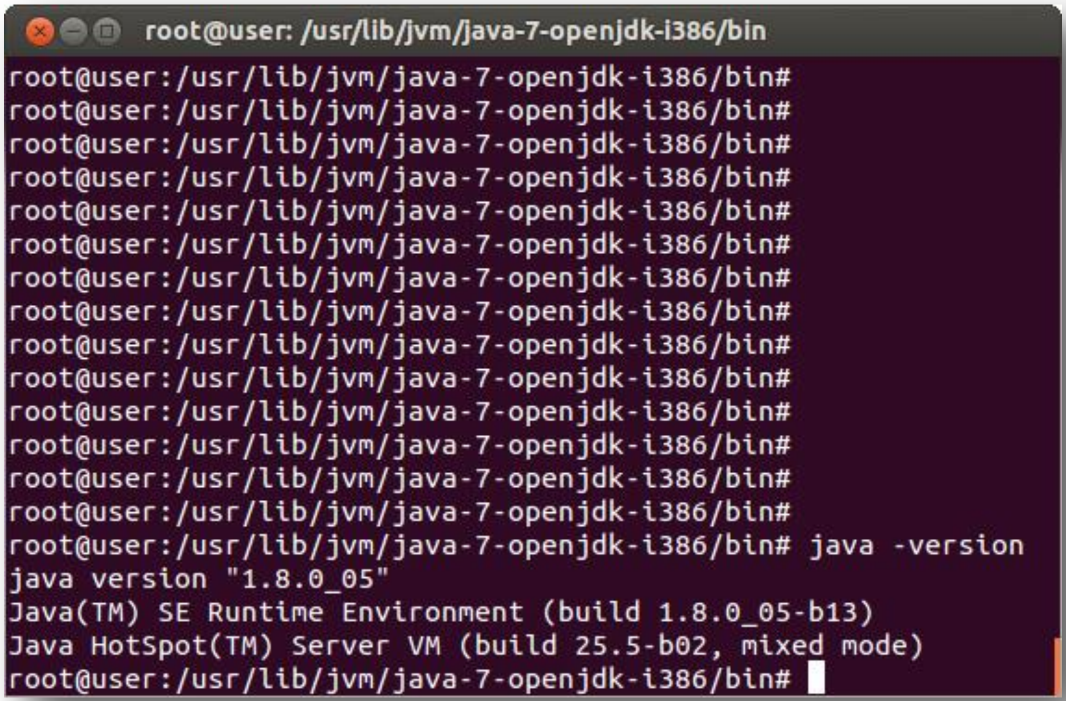
1.2.1. Verification of JRE installation

Check whether JRE is installed on the system or not by running the below command on the command prompt on terminal.

Test your environment by typing:

```
$java –version
```

The commands above will show the current installed version of Java on the system; if JRE is installed then the sample output on Linux terminal would look similar to the following:



root@user: /usr/lib/jvm/java-7-openjdk-i386/bin

```
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin# java -version
java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) Server VM (build 25.5-b02, mixed mode)
root@user:/usr/lib/jvm/java-7-openjdk-i386/bin#
```

If JRE is installed then skip Section 1.2.2 and proceed to Section 1.3, otherwise continue with next section (Section 1.2.2).

1.2.2. Install JRE

Java 1.6 JRE/JDK or above is recommended for Eclipse 4.3 installation, and Java 1.5 JRE/JDK or above is recommended for Eclipse 3.5 installation.

Download the JRE/JDK Installation executable based on platform form the link – http://www.java.com/en/download/manual.jsp, and follow installation instructions from the link – http://www.java.com/en/download/help/download_options.xml.

Note: The default path should be like, *$/usr/lib/jvm/java-6-openjdk-i386/jre*

1.3. Eclipse Installation

SHP-SDK supports Eclipse installation of version 3.5 (code name: **Galileo**) onwards, however, take note that all the screenshots and references used and/or mentioned in this manual are taken from Eclipse installation 4.3 (code name: **Kepler**) and above.

1.3.1. Verification of Eclipse installation

There is no direct way to find whether Eclipse is installed on a system or not.

One of the possible ways to find out on a Linux system (it works only for the installation which comes with Linux) is:

```
$ file /usr/bin/eclipse
eclipse: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.2.5, not stripped
```

Note: Eclipse also allows more than one installation on same system

If Eclipse is already installed then proceed to Section 1.4, otherwise continue with Section 1.3.2

1.3.2. Install Eclipse

Download Eclipse from http://www.eclipse.org/downloads/ and unzip to a local directory.

Eclipse can be launched through the terminal from the downloaded folder.

1.4. Eclipse CDT Plug-in Installation

1.4.1. Verification of Eclipse CDT Plug-in installation

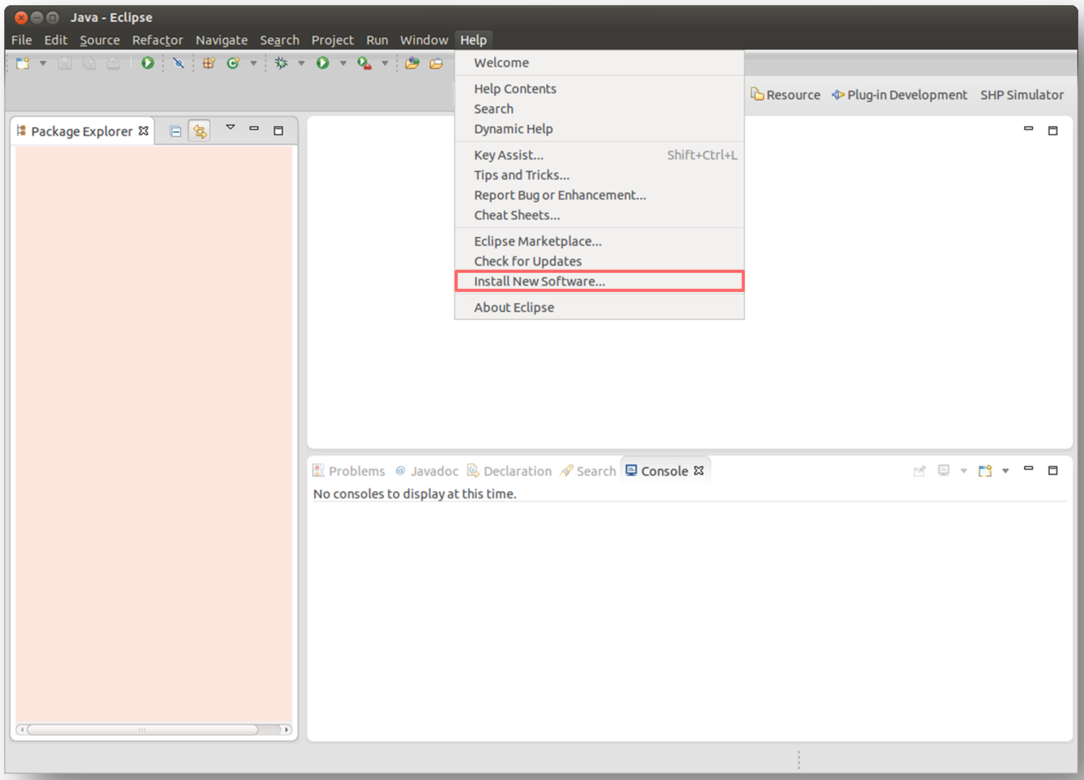
Check whether the CDT Plugin is installed on Eclipse installation or not. Go to Help → About Eclipse SDK → Installation Details → Features. In the 'Feature Id' column check whether 'org.eclipse.cdt' is present. If present then it means CDT has already been installed, then skip Section 1.4.2 and proceed to Section 1.5, otherwise continue with Section 1.4.2.

NOTE: Application development using Eclipse require presence of a compiler, for example, GCC compiler for Linux. Instructions for respective compiler installation are provided in Section 1.5 (Compiler Installation).

1.4.2. Install Eclipse CDT Plug-in

Following are the sequence of steps for installation of Eclipse CDT plug-in:

Step 1: Launch Eclipse ⇒ Help ⇒ Install New Software

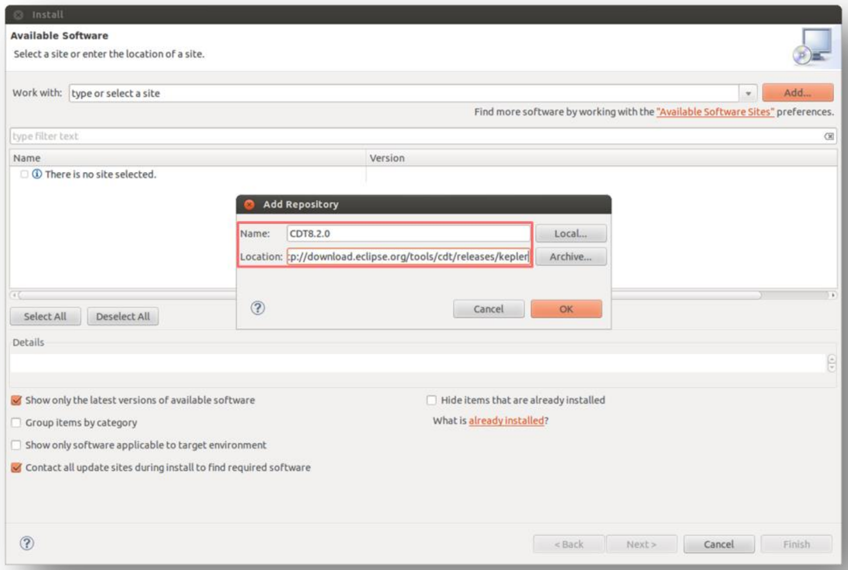


No consoles to display at this time.

Step 2: In "Add", enter location as the CDT update site

http://download.eclipse.org/tools/cdt/releases/kepler (for Eclipse Kepler only) and enter a name (e.g., CDT 8.2.0)

NOTE: The link above will change based on the installation of Eclipse, for different versions of Eclipse installations, the link will change with the release name at the last.


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

There is no site selected.

Add Repository

Name: CDT8.2.0

Location: p://download.eclipse.org/tools/cdt/releases/kepler

[✓] Show only the latest versions of available software

[ ] Group items by category

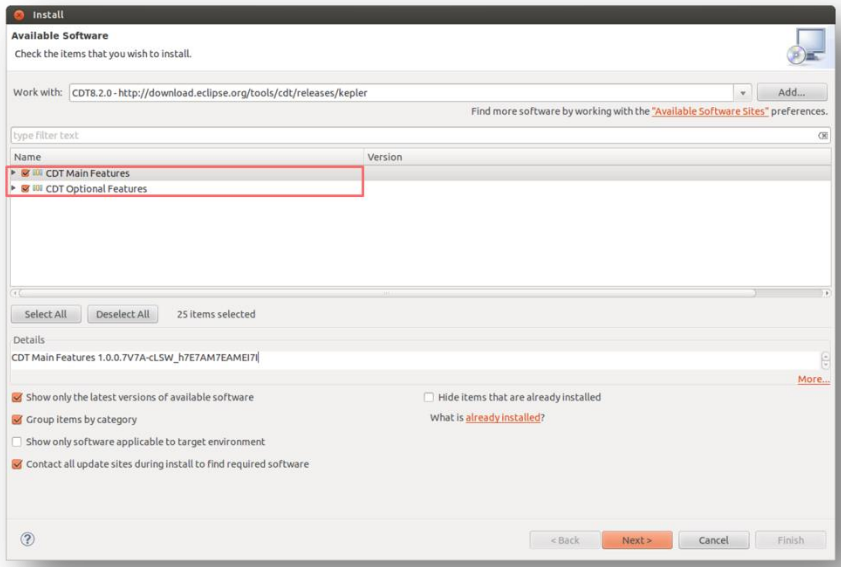
[ ] Show only software applicable to target environment

[✓] Contact all update sites during install to find required software

[ ] Hide items that are already installed

What is already installed?

Step 3: Optional features of CDT can also be selected however, it is mandatory to select all main features of CDT, ("CDT Main Features").


Install

Available Software

Check the items that you wish to install.

Work with: CDT8.2.0 - http://download.eclipse.org/tools/cdt/releases/kepler

Find more software by working with the "Available Software Sites" preferences.

[✓] CDT Main Features

[✓] CDT Optional Features

25 items selected

Details: CDT Main Features 1.0.0.7V7A-cLSW_h7E7AM7EAMEI7I

[✓] Show only the latest versions of available software

[✓] Group items by category

[ ] Show only software applicable to target environment

[✓] Contact all update sites during install to find required software

[ ] Hide items that are already installed

What is already installed?

1.5. Compiler and Database Installation

1.5.1. Install *gcc* compiler for Linux

A) Download the ‘gcc-4.8.1.tar.gz’ for gcc from ftp://ftp.gnu.org/gnu/gcc/gcc-4.8.1/

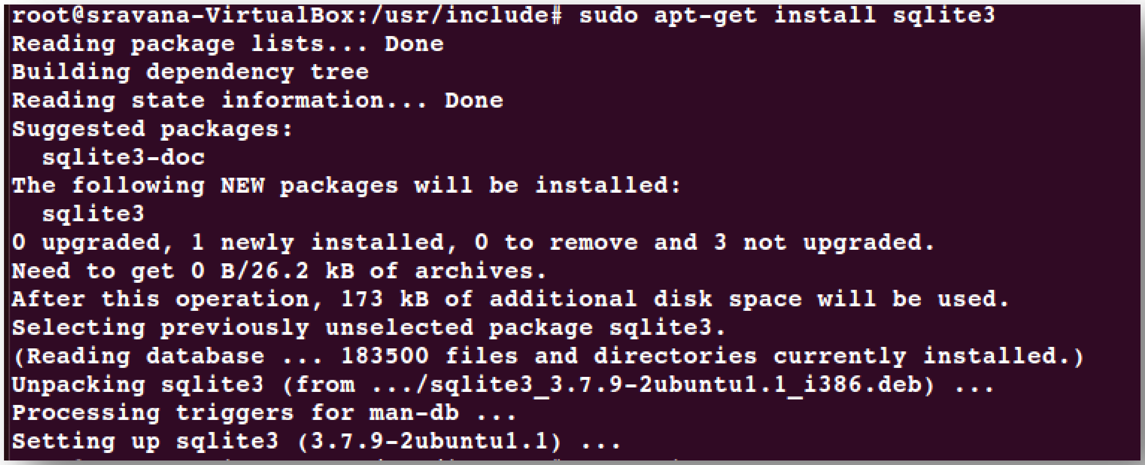
B) Verify the gcc installation by listing the version of gcc :

$gcc –version

1.5.2. Install *SQLite3* for Linux

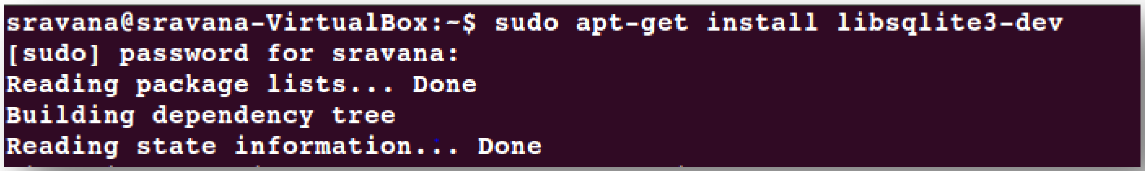
Installation of SQLite3 onto respective development environment is essential for developing applications involving database operations.

- If ‘sqlite3’ is not installed then install it in the following way:



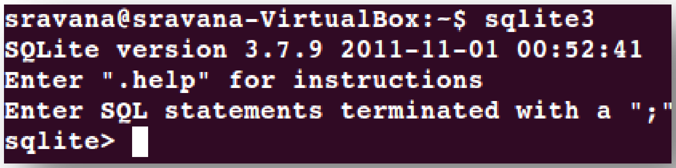
```
root@sravana-VirtualBox:/usr/include# sudo apt-get install sqlite3
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  sqlite3-doc
The following NEW packages will be installed:
  sqlite3
0 upgraded, 1 newly installed, 0 to remove and 3 not upgraded.
Need to get 0 B/26.2 kB of archives.
After this operation, 173 kB of additional disk space will be used.
Selecting previously unselected package sqlite3.
(Reading database ... 183500 files and directories currently installed.)
Unpacking sqlite3 (from .../sqlite3_3.7.9-2ubuntu1.1_i386.deb) ...
Processing triggers for man-db ...
Setting up sqlite3 (3.7.9-2ubuntu1.1) ...
```

- If ‘libsqlite3-dev’ is not installed then install in the following way:



```
sravana@sravana-VirtualBox:~$ sudo apt-get install libsqlite3-dev
[sudo] password for sravana:
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

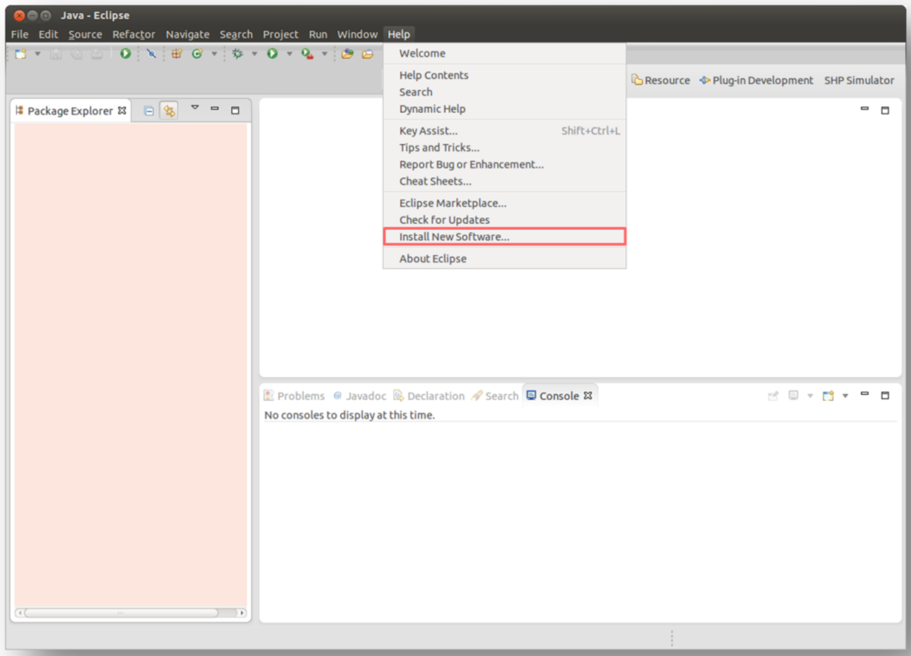
- Verify installation of ‘sqlite3’ in the following way:



```
sravana@sravana-VirtualBox:~$ sqlite3
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

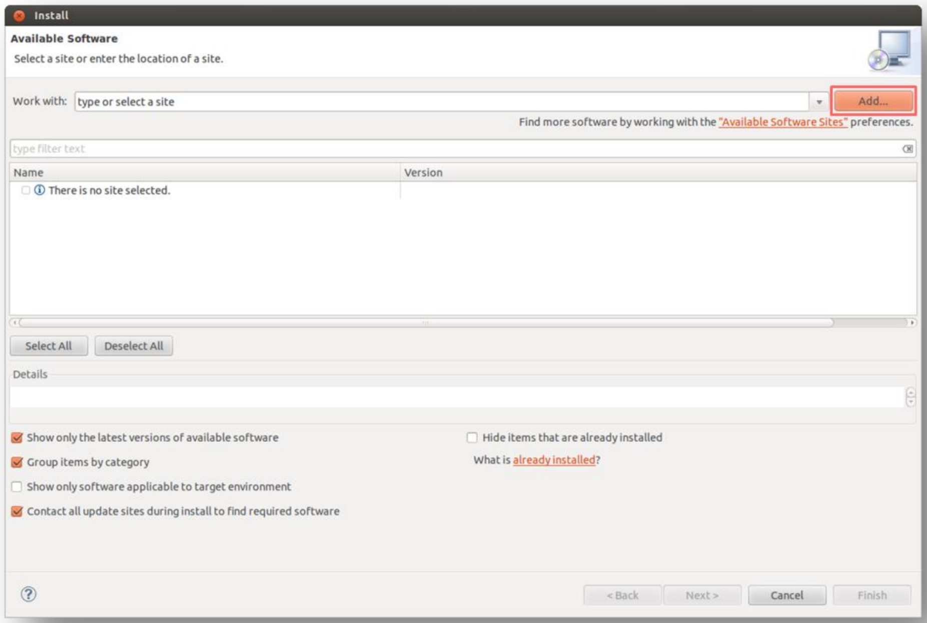
1.6. ‘SHP-SDK’ Plugin Installation

Step 1: Launch Eclipse, click on ‘Help’ → ‘Install New Software’



No consoles to display at this time.

Step 2: Click on ‘Add’



Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

type filter text

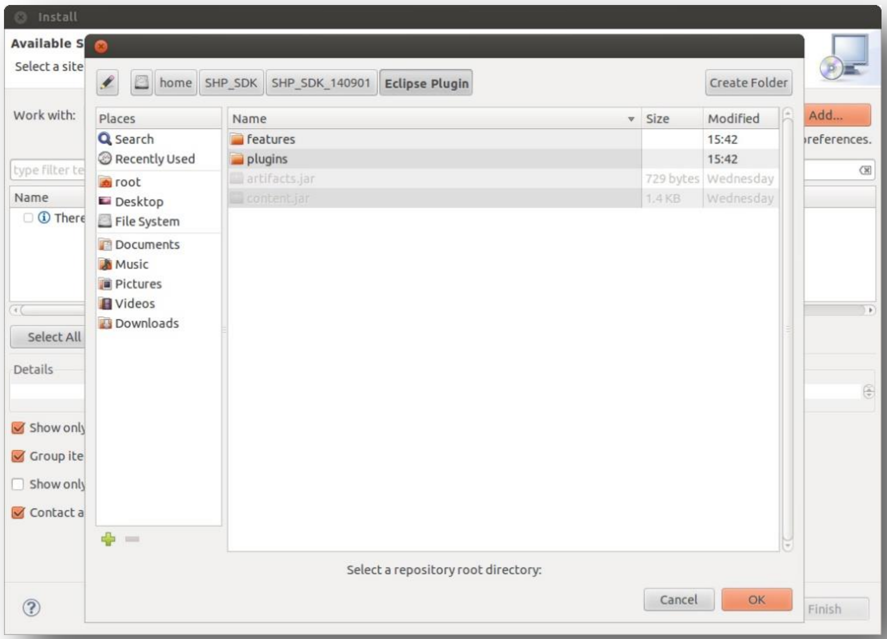
Name	Version
There is no site selected.	

Details

- [✓] Show only the latest versions of available software
- [✓] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software
- [ ] Hide items that are already installed

What is already installed?

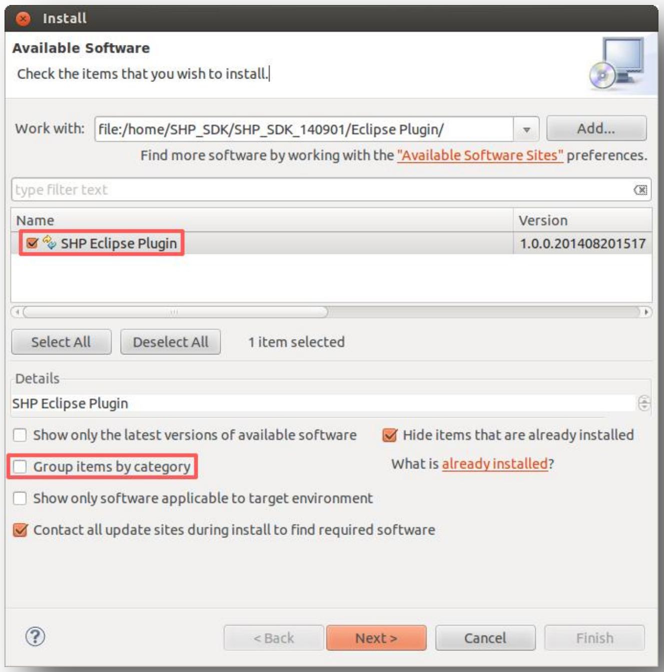
Step 3: Select ‘Local’ and select SHP-SDK plugin folder (from the local location) and click on ‘OK’: ‘*<Downloaded Location of Plugin>*/**EclipsePlugin**’, on



Name	Size	Modified
features		15:42
plugins		15:42
artifacts.jar	729 bytes	Wednesday
content.jar	1.4 KB	Wednesday

Select a repository root directory:

Step 4: Now select ‘SHP Eclipse Plugin’, unselect option ‘Group items by category’ under ‘Details’ section, and click on ‘Next’



Check the items that you wish to install.

Work with: file:/home/SHP_SDK/SHP_SDK_140901/Eclipse Plugin/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
✓ SHP Eclipse Plugin	1.0.0.201408201517

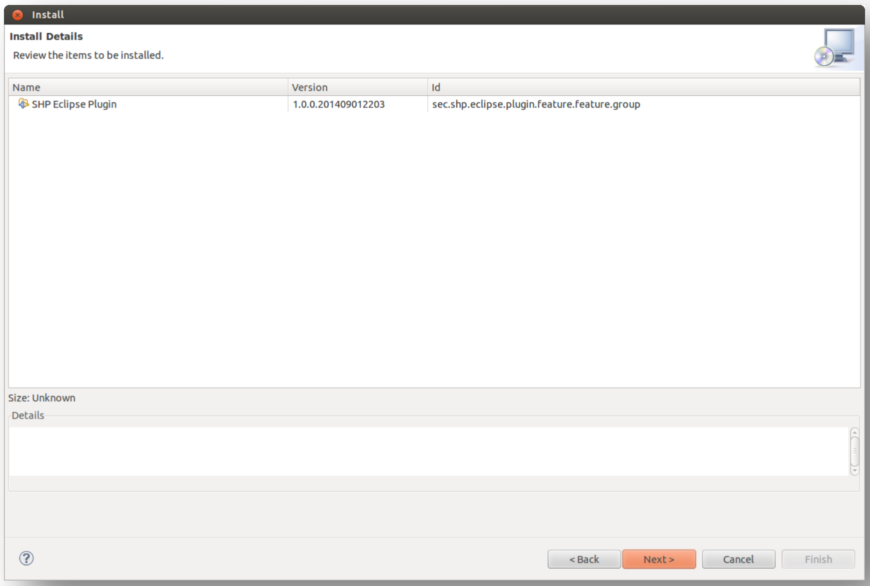
1 item selected

Details

SHP Eclipse Plugin

- [ ] Show only the latest versions of available software
- [✓] Hide items that are already installed
- [ ] Group items by category
- What is already installed?
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software

Step 5: Click ‘Next’ and ‘Finish’



Install Details

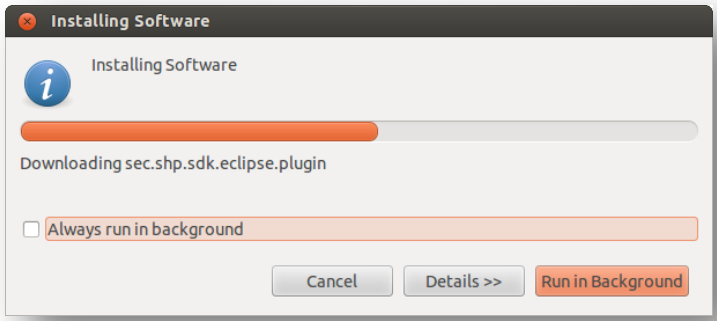
Review the items to be installed.

Name	Version	Id
SHP Eclipse Plugin	1.0.0.201409012203	sec.shp.eclipse.plugin.feature.feature.group

Size: Unknown

Details

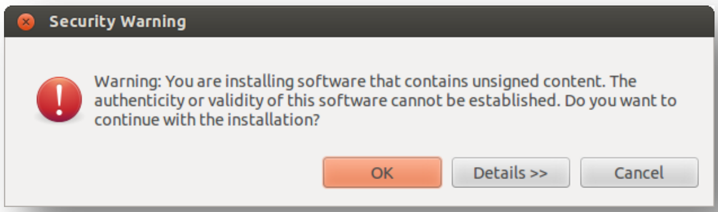
Step 6: This will install SHP-SDK-Plugin onto local system



Installing Software

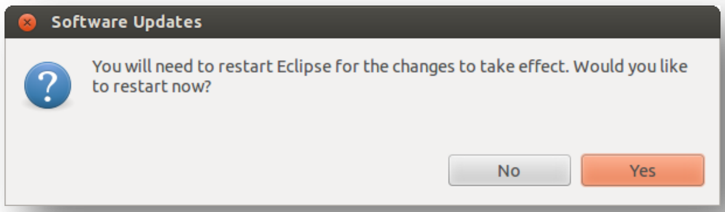
Downloading sec.shp.sdk.eclipse.plugin

Step 7: Approve all security warnings by clicking on ‘OK’:



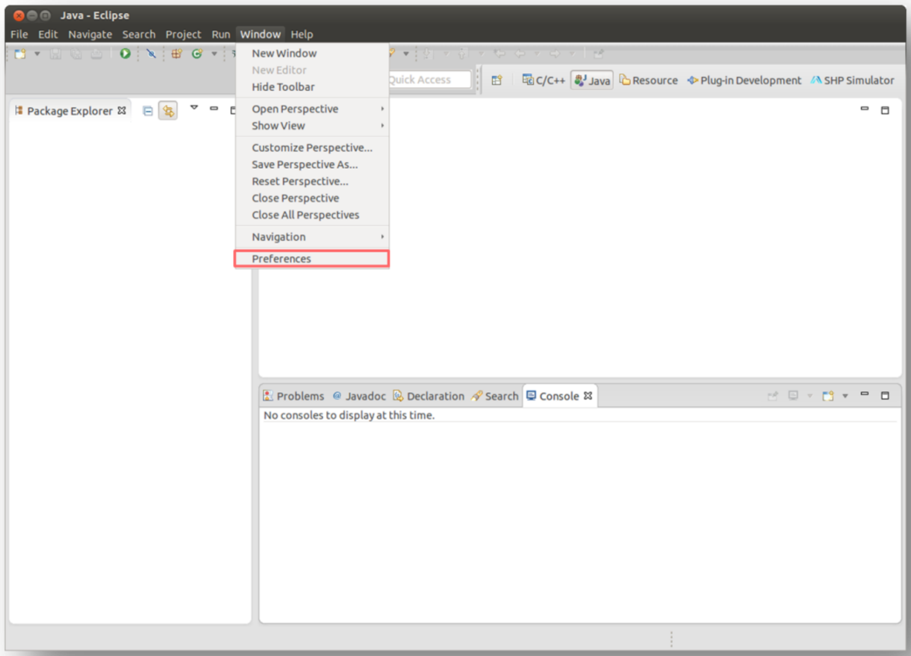
Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

Step 8: Click on 'Yes' on 'Eclipse Restart' confirmation window, close the Eclipse window, and proceed to next step:


Software Updates

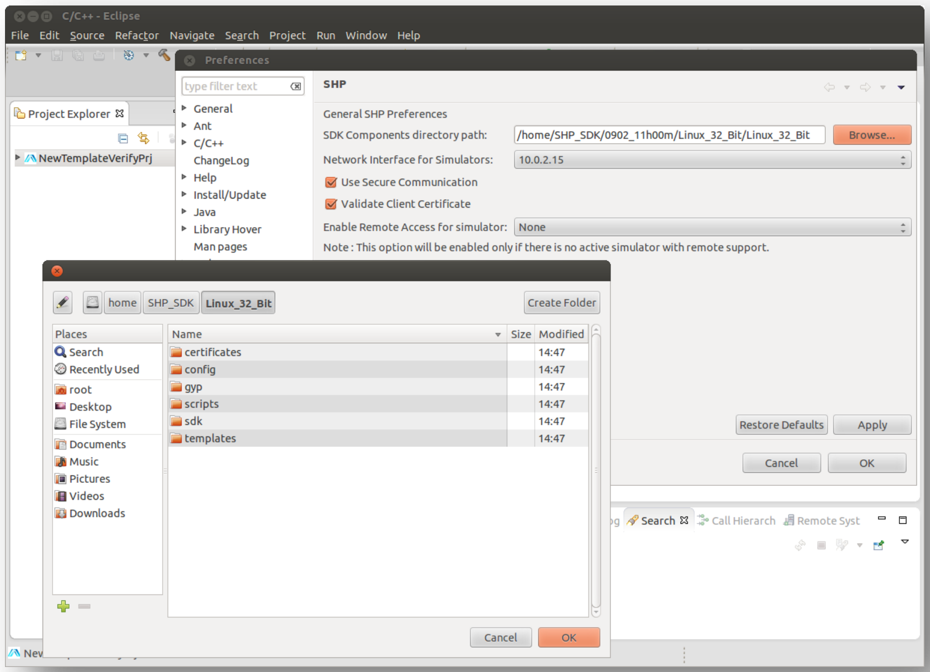
You will need to restart Eclipse for the changes to take effect. Would you like to restart now?

Step 9: Once Eclipse has restarted successfully then we need to change SHP preferences, select 'Window → Preferences':



No consoles to display at this time.

Step 10: On this, we need to setup SHP preferences by setting 'SDK Components directory path' to '<*Downloaded Location of SHP-SDK* >/ Linux_32_Bit/ ' location for SHP-SDK-Eclipse Plugin installation:



SHP

General SHP Preferences

SDK Components directory path: /home/SHP_SDK/0902_11h00m/Linux_32_Bit/Linux_32_Bit

Network Interface for Simulators: 10.0.2.15

✓ Use Secure Communication

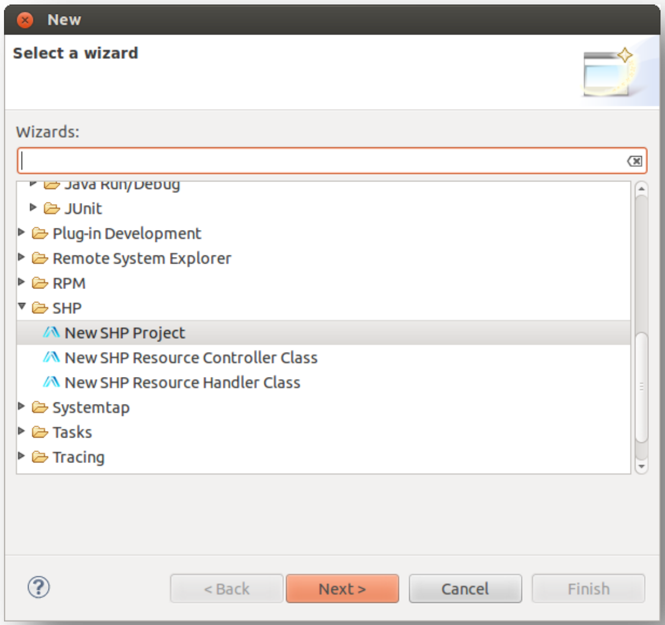
✓ Validate Client Certificate

Enable Remote Access for simulator: None

Note : This option will be enabled only if there is no active simulator with remote support.

Name	Size	Modified
certificates		14:47
config		14:47
gyp		14:47
scripts		14:47
sdk		14:47
templates		14:47

Step 11: This finishes 'SHP-SDK-Eclipse plugin' installation, now launch 'Eclipse', and we shall be able to create new SHP Projects, try to create new SHP project (File → New → Other), and shall see that options to create SHP projects are available as below:

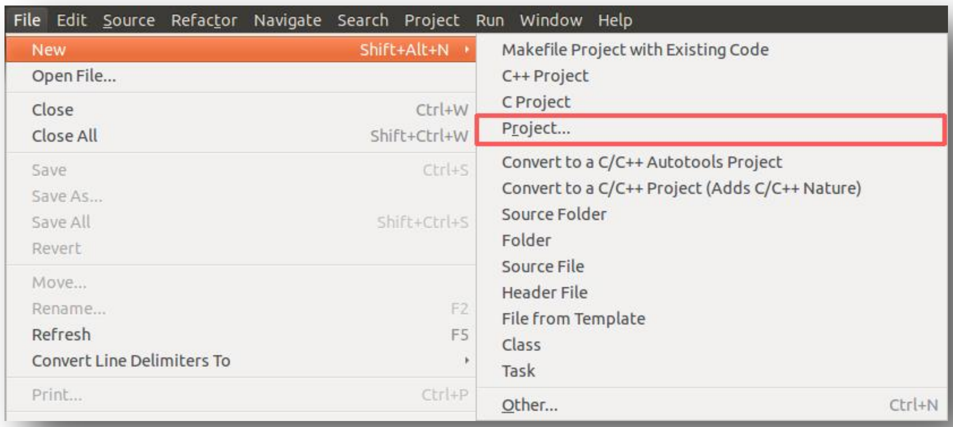


2. Application Development

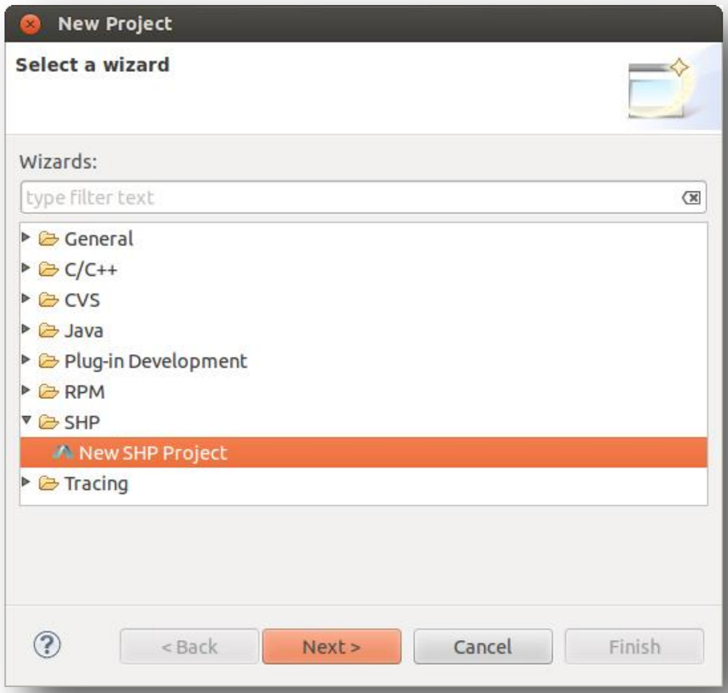
2.1. New SHP Project

2.1.1. Creating a SHP Project

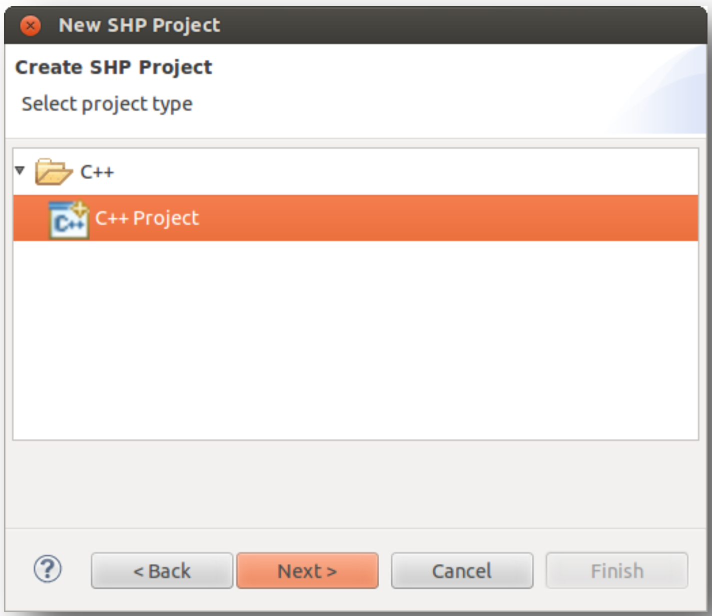
- Select 'File → New → Project'

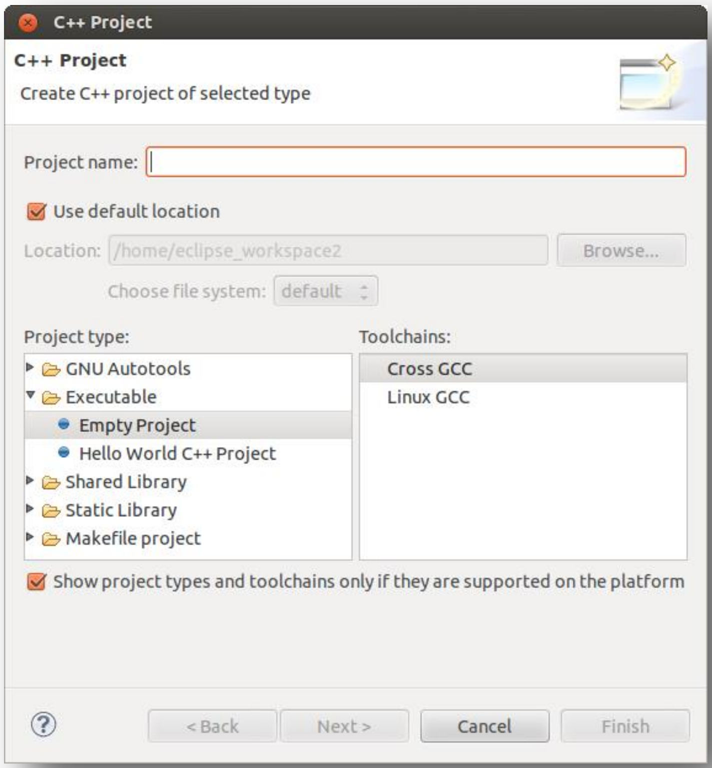


- Select 'SHP → New SHP Project'



- Select type of project which needs to be created (C++), SHP project wizard will invoke selected project wizard.
- Create a new project by giving a name to the project and select appropriate tool chain.

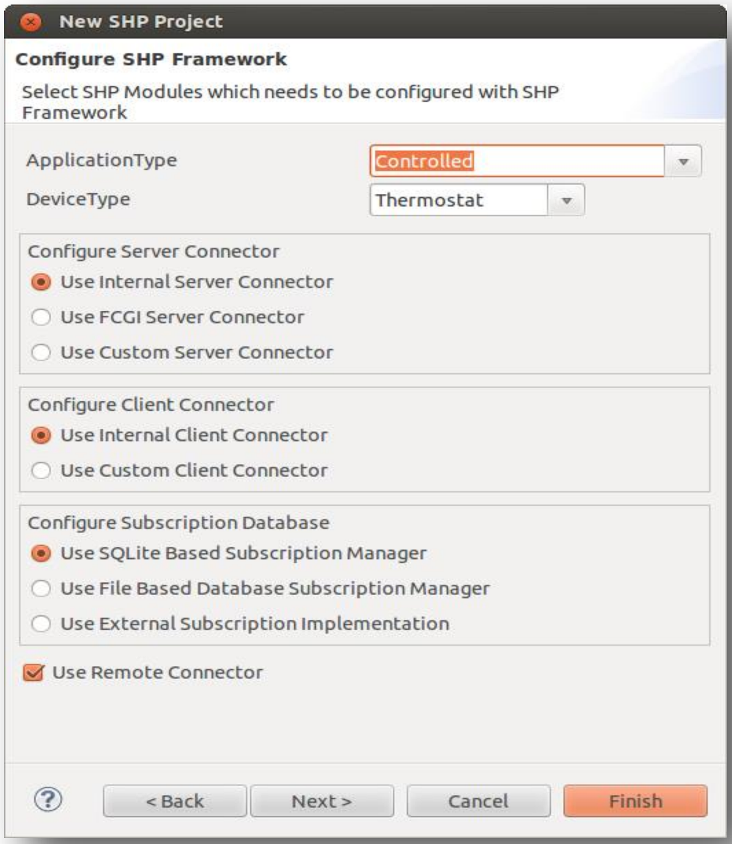




2.1.2. Configuring a SHP project

‘Configuration SHP Project’ enables SHP-Application developers to configure (type of device, Server/Client connectors to be used) SHP-Framework (their projects) based on their needs.

Following are different configuration options provided by SHP-SDK for application development:



New SHP Project

Configure SHP Framework

Select SHP Modules which needs to be configured with SHP Framework

ApplicationType: Controlled

DeviceType: Thermostat

Configure Server Connector
- [✓ Use Internal Server Connector]
- Use FCGI Server Connector
- Use Custom Server Connector

Configure Client Connector
- [✓ Use Internal Client Connector]
- Use Custom Client Connector

Configure Subscription Database
- [✓ Use SQLite Based Subscription Manager]
- Use File Based Database Subscription Manager
- Use External Subscription Implementation

[✓ Use Remote Connector]

Respective explanation for each Configuration option is as follows:

- **Application Type:** Using this, developers can select type of application to be created, select ‘Controlled’ from the drop down
  - **Controlled:** Controlled by other devices (e.g. Washer, Refrigerator, Thermostat, LED, Smart Plug)
- **Device Type:** Using this, developers can select the type of device for which the application is being created, developers can select any one of the device types supported by SHP
  - Based on selected device type SHP-SDK makes some automatic suggestion of Resources (Section 2.1.3) to be Controlled
- **Configure Server Connector:** Using this, developers can select kind of Server Connector to be used for the application being created.
  - SHP-Framework provides two types of server connectors, one is ‘Internal Server Connector’ which is in-house implementation (this is the default selection – ‘Use Internal Server Connector’), and the other one is FCGI based (for this select ‘Use FCGI Server Connector’) server connector
    - ➢ Developers can also develop their own (custom) Server Connector by implementing `Sec::Shp::Connector::Server::IServerConnector` interface

- **Configure Client Connector:** Using this, developers can select kind of Client Connector to be used for the application being created.
  - SHP-Framework provides, one 'Internal Client Connector' which is in-house implementation (this is the default selection – 'Use Internal Client Connector')
    - ➢ Developers can also develop their own (custom) Client Connectors by implementing `Sec::Shp::Connector::Client::IClientConnector` interface
- **Configure Subscription Database** (subscription implementation)**:** Using this, developers can select kind of Subscription feature implementation.
  - SHP-Framework provides two types of subscription implementations, one is using 'SQLite database' (this is the default selection – 'Use SQLite Subscription Manager'), and the other one is file based (for this select 'Use File Based Database Subscription Manager') implementation
    - ➢ Developers can also develop their own (custom) subscription implementation by implementing `Sec::Shp::Notification::ISubscriptionManager` interface
- **Use Remote Connector:** Developers needs to select (by default selected) this feature if the application being developed needs to have Remote Access feature of SHP, otherwise, they are expected to unselect this option

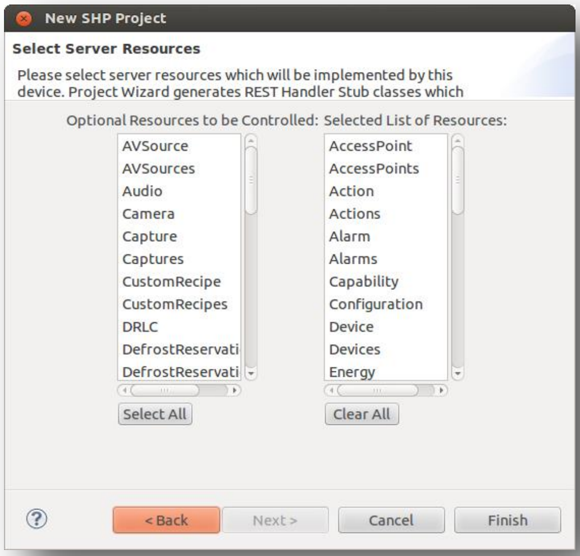
Note: Selection of **custom** connectors (Server/Client) expects application developers to have their own implementation for REST message handling (construction and parsing)

2.1.3. Selecting Resources

Selection of ***Controlled*** application type, leads to **'Selection of Server Resources**' (resources to be handled by the *Controlled* application) screen. **Mandatory Resources** are selected automatically. Mandatory Resources **cannot be removed**

- On Selecting a Device Type, the suggested Resources are automatically selected (will be automatically moved) to 'Selected List of Resources:'
- Other Resources can be added or removed as per requirement by selecting (click) from 'Optional Resources to be Controlled:'
- Remove unwanted resources (non-mandatory) by clicking on them in the Selected Resources List.

Note: These resources are characteristic of a controlled device, which acts as a server and responds to the requests it receives.


New SHP Project

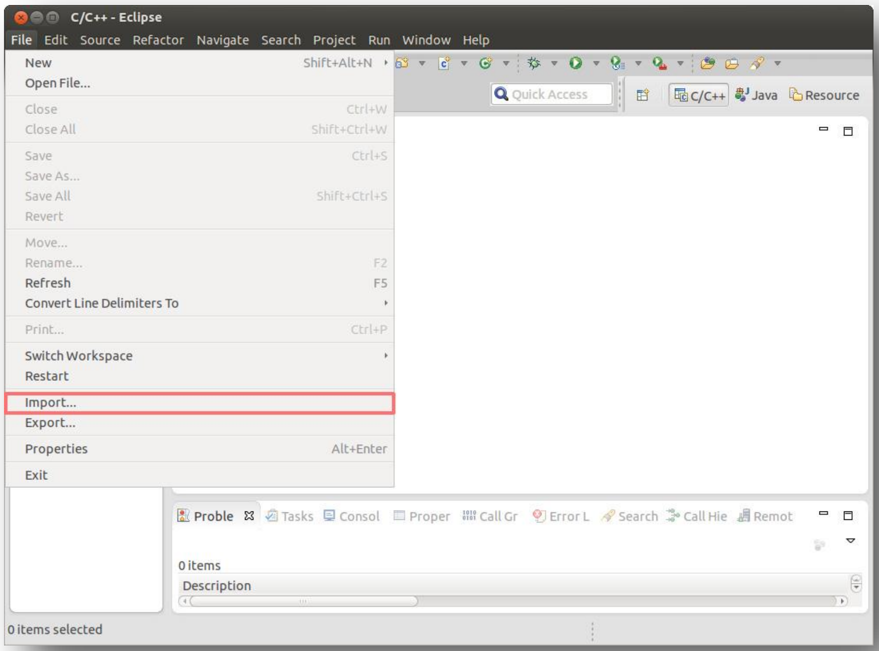
Select Server Resources

Please select server resources which will be implemented by this device. Project Wizard generates REST Handler Stub classes which

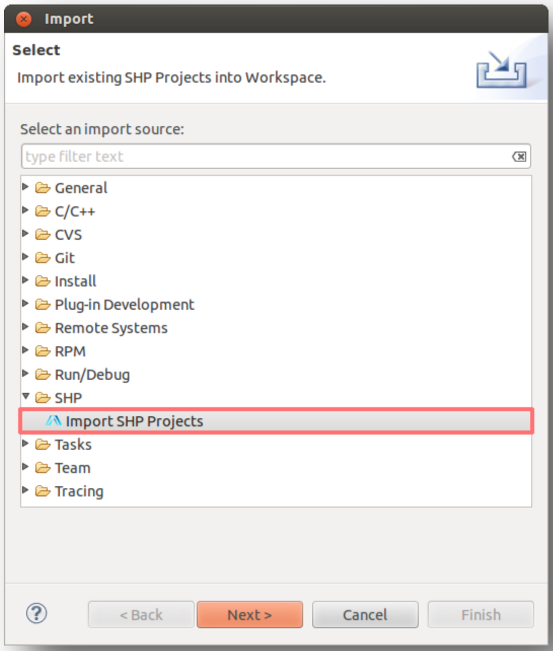
Optional Resources to be Controlled:	Selected List of Resources:
AVSource	AccessPoint
AVSources	AccessPoints
Audio	Action
Camera	Actions
Capture	Alarm
Captures	Alarms
CustomRecipe	Capability
CustomRecipes	Configuration
DRLC	Device
DefrostReservati	Devices
DefrostReservati	Energy

2.2. Import existing SHP Projects into the Eclipse workspace

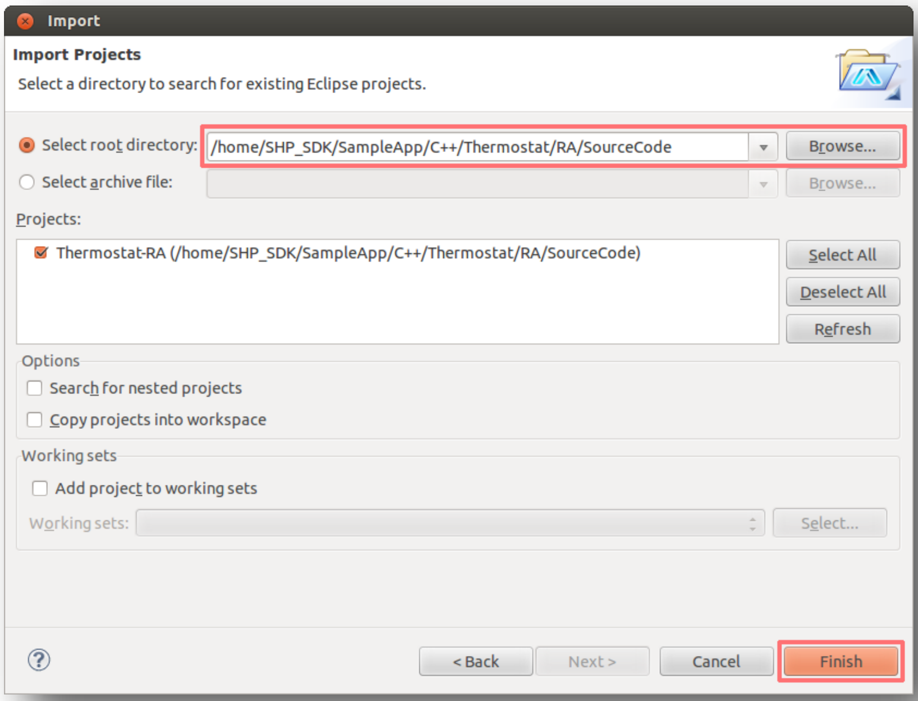
1. Select 'File → Import'



2. Select 'Select an import source → SHP → Import SHP Projects → Next'



3. Select ‘Select root directory(Where source code locates) → Browse… → Finish’


Import

Import Projects

Select a directory to search for existing Eclipse projects.

● Select root directory: /home/SHP_SDK/SampleApp/C++/Thermostat/RA/SourceCode

○ Select archive file:

Projects:

☑ Thermostat-RA (/home/SHP_SDK/SampleApp/C++/Thermostat/RA/SourceCode)

Options

☐ Search for nested projects

☐ Copy projects into workspace

Working sets

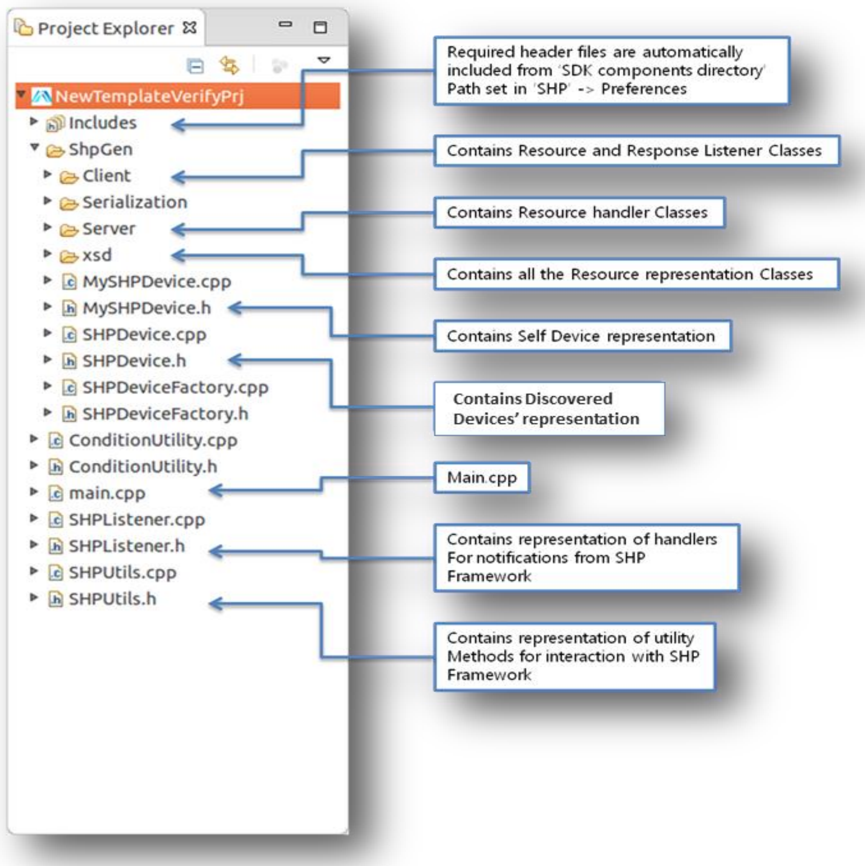
☐ Add project to working sets

Working sets:

2.3. Application Project

On Creating a New Project, the SHP Code Generator automatically generates following **folder structure**:

- For a C++ Project:

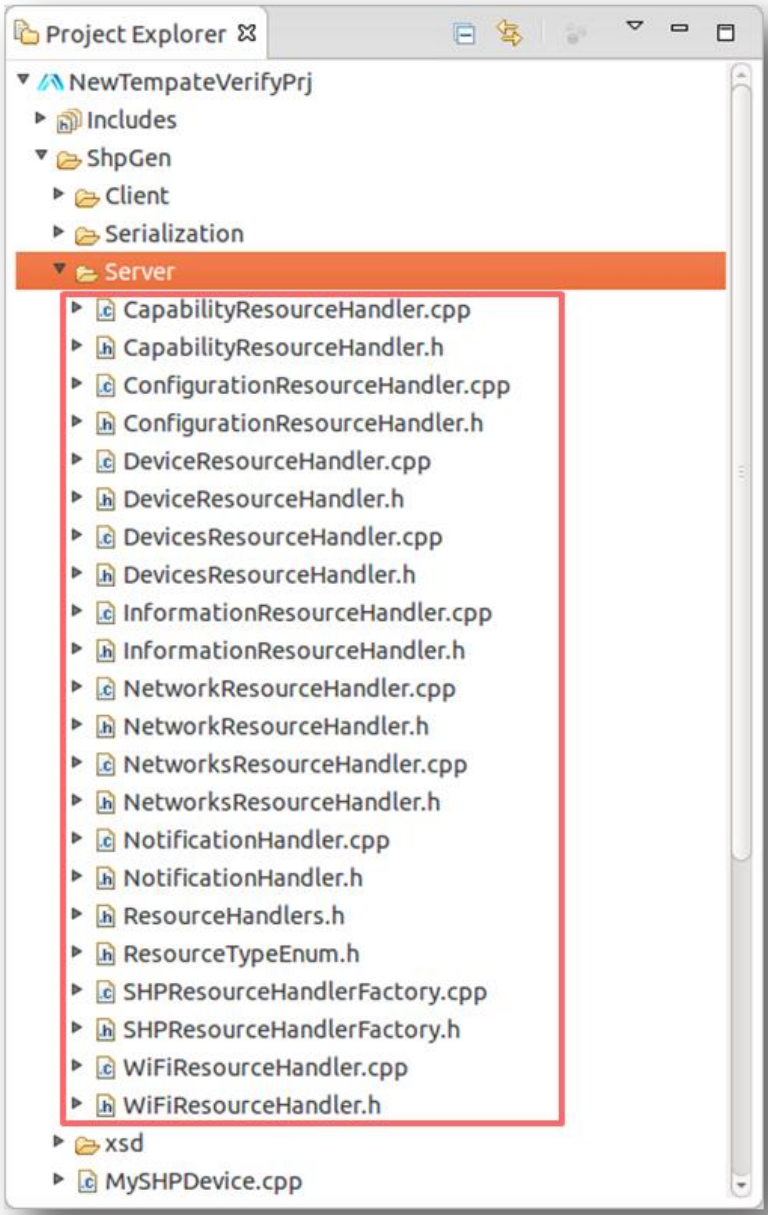

Project Explorer
NewTemplateVerifyPrj
Includes
ShpGen
Client
Serialization
Server
xsd
MySHPDevice.cpp
MySHPDevice.h
SHPDevice.cpp
SHPDevice.h
SHPDeviceFactory.cpp
SHPDeviceFactory.h
ConditionUtility.cpp
ConditionUtility.h
main.cpp
SHPListener.cpp
SHPListener.h
SHPUtils.cpp
SHPUtils.h
Required header files are automatically included from 'SDK components directory' Path set in 'SHP' -> Preferences
Contains Resource and Response Listener Classes
Contains Resource handler Classes
Contains all the Resource representation Classes
Contains Self Device representation
Contains Discovered Devices' representation
Main.cpp
Contains representation of handlers For notifications from SHP Framework
Contains representation of utility Methods for interaction with SHP Framework

- **'*Server'* folder in C++ Project** contains Resource handler classes that are called when a particular request (allowed SHP-REST request) is received on a specific Resource, these are to be developed.
- The Application developer should ensure that the Server responds properly to valid (allowed as per SHP specification) requests for each Resource.
  - ***This can be ensured by developing all the required Resource Handler methods left out for the developer (check the TODO list on Eclipse).***
- The **'*XSD* folder in C++ Project** contains classes corresponding to various Resources.
- A few of the members of these classes are optional members while some are mandatory members.
- If the mandatory members are not dealt with properly (setting or getting their values depending on whether sending a request or sending a response), the serialization (or deserialization fails).
- The Serialization folder contains methods to serialize and de-serialize the Serializable data.
- Additionally, ***'SHPUtils.cpp'* class** contains methods to initialize and also to start the framework apart from other methods required to configure it. And ***'SHPListener.cpp' class*** in C++ project contains representation of handlers for notifications from SHP Framework

2.3.1. Server Classes

Developers need to develop the resource handler stubs which are generated by SHP Plug-in for each of the selected resources while project creation.

This is a characteristic of a ***Controlled*** device, which responds to a REST request received for a particular Resource. This response should be set in the Resource handler file, for each of the valid request methods.



```
Project Explorer
▼ NewTempateVerifyPrj
  ▶ Includes
  ▼ ShpGen
    ▶ Client
    ▶ Serialization
    ▼ Server
      ▶ CapabilityResourceHandler.cpp
      ▶ CapabilityResourceHandler.h
      ▶ ConfigurationResourceHandler.cpp
      ▶ ConfigurationResourceHandler.h
      ▶ DeviceResourceHandler.cpp
      ▶ DeviceResourceHandler.h
      ▶ DevicesResourceHandler.cpp
      ▶ DevicesResourceHandler.h
      ▶ InformationResourceHandler.cpp
      ▶ InformationResourceHandler.h
      ▶ NetworkResourceHandler.cpp
      ▶ NetworkResourceHandler.h
      ▶ NetworksResourceHandler.cpp
      ▶ NetworksResourceHandler.h
      ▶ NotificationHandler.cpp
      ▶ NotificationHandler.h
      ▶ ResourceHandlers.h
      ▶ ResourceTypeEnum.h
      ▶ SHPResourceHandlerFactory.cpp
      ▶ SHPResourceHandlerFactory.h
      ▶ WiFiResourceHandler.cpp
      ▶ WiFiResourceHandler.h
    ▶ xsd
    ▶ MySHPDevice.cpp
```

2.4. Developing Application

Developing SHP applications majorly consists of following steps:

1. **Initialization of SHP Framework**
    a. Configuration of required certificates
        i. Setting certificates path for Server and Client (**Please note that default certificate provided in this package can be used for the testing purposes only. In order to apply to the commercial product, new model certificate shall be issued from the Samsung Electronics.**)
        ii. Setting Remote Access configuration file path (**If remote access features is required by the application**)
    b. Configuration of Self Device Details – for setting device specific details
        i. Like IP Address, Port, UUID, device type, application type, device information related details and etc.
        ii. Setting of supported resources – resources to be controlled (for *Controller* application) or handled (*Controlled* application)
    c. Configuration of Subscription Manager – for handling subscriptions and receiving notification
    d. Configuration of SHP Connectors
        i. Server Connectors – for handling and serving requests from SHP Controller devices
        ii. Client Connectors – for retrieving SHP device details and sending control commands
    e. Configuration related to remote access (**If remote access features is required by the application**)
        i. Configuration and initialization of all servers (Samsung Account Server, Smart Home Server – SHS, and SCS) involved for remote access
    f. Initialization of required factories – for example, device, serialization, and resource handler
    g. Setting and verification of final configuration
2. **Creation, initialization, and setting of SHP Framework listeners**
    a. Creation and configuration of Device Finder Listener – for handling SHP Devices related notifications, for example, new SHP device discovery, modification to existing SHP device, and device leaving network
    b. Creation and configuration of SHP Status Listener – for handling notifications from SHP Framework related status (start/ stop / error) of framework, easy setup, registration, and device token related
3. **Starting SHP Framework**
    a. Check whether application (running on device) is provisioned or not?
        i. Check whether application possess all required details to be connected to Home Access Point (Home AP) for registering onto cloud OR not?
            1. If application is not provisioned then enable Soft AP mode onto device and perform Easy Setup routine (refer to SHP Architecture for complete details on Easy Setup routine)
4. **Discovering devices**
    a. Implementation of *IDeviceFinderListener*
    b. Registering *DeviceFinderListener*
    c. Retrieving Discovered devices
5. **Performing Resource Control / Monitor / Manipulation**
    a. Sending REST requests to discovered devices
    b. Handling REST requests from controller devices
6. **Stopping SHP Framework**
    a. Remove / Un-set all listeners
    b. Rest Configuration, and
    c. Cleanup memory
7. **Easy Setup, Registration, and Remote Access**

2.4.1. Initialization of SHP Framework for C++ Controlled App

`SHPUtils::initializeFramework()` of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp** is the function which is responsible for complete initialization of SHP Framework.

2.4.1.1. Configuration of required Certificates

By default, Server and Client security related file paths are set using "SDK Components directory path" preference field value of SHP Preferences Page (*Window* → *Preferences* → *SHP*). And all certificates and certificate's key files are stored in certificates folder under "SDK Components directory path" preference field value.

The following are macros used for setting security related file paths and passphrase. The application developer needs to modify these macros case of storing security related files in different folder or used different files.

SERVER_ROOT_CA: This macro is used to specify root certificate or chain of root certificates file path, which is used for issuing server certificate. Default file path is
<"SDK Components directory path" preference field value>/certificates/Standalone_CA.crt"

SERVER_SELF_CERTIFICATE_RSA: This macro is used to specify server certificate file path. Default file path is <"SDK Components directory path" preference field value >/certificates\Server.crt"

SERVER_SELF_CERTIFICATE_PRIVATE_KEY_PATH: This macro is used to specify server certificate's private key file path. Default file path is
<"SDK Components directory path" preference field value >/certificates/Server.pem"

SERVER_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD: This macro is used to specify server certificate's private key passphrase. Default value is "SHPSDK_SERVER_TEST_CERTIFICATE".

CLIENT_ROOT_CA: This macro is used to specify root certificate or chain of root certificates file path, which is used for issuing client certificate. Default file path is
<"SDK Components directory path" preference field value>/certificates/Standalone_CA.crt"

CLIENT_SELF_CERTIFICATE_RSA: This macro is used to specify client certificate file path. Default file path is <"SDK Components directory path" preference field value >/certificates\Client.crt"

CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_PATH: This macro is used to specify client certificate's private key file path. Default file path is
<"SDK Components directory path" preference field value >/certificates/Client.pem"

CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD: This macro is used to specify client certificate's private key passphrase. Default value is "SHPSDK_CLIENT_TEST_CERTIFICATE".

COMMON_ROOT_CA: This macro is used to specify chain of root certificates file path, which is used for issuing Service Server and Account Server Certificates.

Following is the code snipped from file : ***<SHP-C++_Project_Name>*\SHPUtils.cpp,** which sets up all certificates:

```
#define SERVER_ROOT_CA
"/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Standalone_CA.crt"
/**< Represents Server Root CA Path */
#define SERVER_SELF_CERTIFICATE_RSA "/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Server.crt"
/**< Represents Server RSA version of Self Certificate */
#define SERVER_SELF_CERTIFICATE_PRIVATE_KEY_PATH
"/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Server.pem"
/**< Represents Private Key path for Server Self Certificate */
#define SERVER_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD "SHPSDK_SERVER_TEST_CERTIFICATE"
/**< Represents Private Key password for Server Self Certificate */
#define CLIENT_ROOT_CA "/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Standalone_CA.crt"
/**< Represents Client Root CA Path */
#define CLIENT_SELF_CERTIFICATE_RSA "/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Client.crt"
/**< Represents Client RSA version of Self Certificate */
#define CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_PATH
"/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/Client.pem"
/**< Represents Private Key path for Client Self Certificate */
#define CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD "SHPSDK_CLIENT_TEST_CERTIFICATE"
/**< Represents Private Key password for Client Self Certificate */
#ifdef REMOTE_ACCESS_SUPPORT
#include "xsd/Network.h"
#define COMMON_ROOT_CA "/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/certificates/CA.crt"
/**< Represents Common Root CA file path */
#define RA_CONFIG_FILE_PATH "/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/config/controllable.ra.config"
/**< Represents Remote Access Configuration file path */
#endif
```

2.4.1.2. Configuration of Self Device Details

2.4.1.2.1. Setting of Application’s Port Number

By default, application’s port number is set to “*8888*”. So the application developer should specify application’s port number by updating port variable’s value in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**.

```
Sec::Shp::Configuration* p_config = sp_shp->getConfiguration();
if (p_config == NULL) {
    std::cout << "Failed to get SHP Configuration" << std::endl;
    return false;
}

if (false == p_config->setAppType(Sec::Shp::ApplicationType_Controllable)) {
    std::cout << "Failed to set SHP Aplication Type" << std::endl;
    return false;
}

/** Configure Network Interface and port to be used */
char deviceAddress[256] = {0x00,};
std::string deviceIPAdress;
std::string port = "8888"; /** @note    Default Port */
std::string uuid;
```

2.4.1.2.2. Setting of Application's IP Address

`Sec::Shp::Configuration::setAddress()` is the API to be used for setting Application Address (*IP_Address:Port*) and

`SHPUtils::getIPAddressAndUUID()` is the method generated for getting interfaces IP Address and UUID using MAC address

By default, the application retrieves interface's IP addresses and chooses the first non-multicast IP address from the IP address list and configure as application's IP Address. If the application developer wants to choose a different IP address, he has to override the "deviceIPAdress" variable's value in *SHPUtils::initializeFramework()* method of file: **<*SHP-C++_Project_Name*>\SHPUtils.cpp**.

```
if (false == SHPUtils::getIPAddressAndUUID(deviceIPAdress, uuid))
{
        cout << "SHPUtils::initializeFramework() => " << "failed to determine network
        interface\n";
        return false;
}


if ((deviceIPAdress.empty()) || (uuid.empty()))
{
        cout << "SHPUtils::initializeFramework() => " << "invalid network interface or uuid\n";
        return false;
}


cout << "SHPUtils::initializeFramework() => " << "Selected IPAddress:" << deviceIPAdress << ";
UUID :" << uuid << std::endl;


sprintf(deviceAddress, "%s:%s", deviceIPAdress.c_str(), port.c_str());
if (false == sp_myDevice->setAddress(deviceAddress)) {
        std::cout << "Failed to set SHP Device Address" << std::endl;
        return false;
}
```

2.4.1.2.3. Setting of device specific details

`Sec::Shp::Device::setDeviceType(), Sec::Shp:: Device::setUUID(), Sec::Shp:: Device::setDescription(), Sec::Shp::Device::setManufacturer(), Sec::Shp::Device::setModelID(), and Sec::Shp::Device::setSerialNumber()` are the APIs to be used for setting device specific details by application

By default, the application sets devices' application specific details like device type, and application type based on details provided during project creation, however, application is expected to set other device specific details like mode ID, description, serial number and etc. by modifying default generated values *SHPUtils::initializeFramework()* method of file: **<*SHP-C++_Project_Name*>\SHPUtils.cpp**.

```
    /** Configure Device Details */
    if (false == sp_myDevice->setDeviceType(Sec::Shp::DeviceType_Thermostat)) {
        std::cout << "Failed to set SHP Device Type" << std::endl;
        return false;
    }
    if (false == sp_myDevice->setUUID(uuid.c_str())) { // Example UUID: "E8113233-9A97-0000-0000-
000000000000"
        std::cout << "Failed to set SHP Device UUID" << std::endl;
        return false;
    }

    if (false == sp_myDevice->setDescription("Description")) {
        std::cout << "Failed to set SHP Device Description" << std::endl;
    }

    if (false == sp_myDevice->setManufacturer("Manufacturer")) {
        std::cout << "Failed to set SHP Device Manufacturer Name" << std::endl;
    }

    // User can additionally specify an optional 'deviceSubType' attribute if 'deviceType' is not
sufficient to define the type of device user want to apply.
    // (e.g., System_Air_Conditioner)
    //if (false == sp_myDevice->setDeviceSubType("DeviceSubType")) {
    //  std::cout << "Failed to set SHP Device Sub-Type" << std::endl;
    //}

    if (false == sp_myDevice->setModelID("Model ID")) {
        std::cout << "Failed to set SHP Device Model ID" << std::endl;
    }

    if (false == sp_myDevice->setSerialNumber("Serial Number")) {
        std::cout << "Failed to set Serial Number" << std::endl;
    }
```

2.4.1.2.4. Setting of supported resources

`Sec::Shp::Device::setSupportedResourceType()` is the API to be used for setting all the supported resources

By default, the application sets devices' supported resources based on selected resources during project creation. Application developers can modify (add/delete) by modifying resource type values in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**.

```
/** Configure Supported Resources */
try {
        sp_myDevice->setSupportedResourceType("AccessPoint");
        sp_myDevice->setSupportedResourceType("AccessPoints");
        sp_myDevice->setSupportedResourceType("Action");
        sp_myDevice->setSupportedResourceType("Actions");
        sp_myDevice->setSupportedResourceType("Alarm");
        sp_myDevice->setSupportedResourceType("Alarms");
        sp_myDevice->setSupportedResourceType("Capability");
        sp_myDevice->setSupportedResourceType("Configuration");
        sp_myDevice->setSupportedResourceType("Device");
        sp_myDevice->setSupportedResourceType("Devices");
} catch(...) {
                std::cout << "Caught Exception" << std::endl; return false;
}
```

2.4.1.2.5. Setting of Subscription manager's Database file path

`Sec::Shp::SHP::setSubscriptionManager()` is the API to be used for setting Subscription manager and `Sec::Shp::Configuration::setSubscriptionDbPath()` is the API to be used for setting Subscription Database file path

By default, the Subscription manager's Database file path is set to "SubscriptionManager.db". The database is used to store subscriptions. If the application developer wants to change the data base file path, specify the new database file path by passing the first argument to *setSubscriptionDbPath()* call of "*Sec::Shp::Configuration*" class in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**.

Note: Application developers can also modify type of subscription store also

```
/**
 * Configure Subscription Manager::@n
 *      Applications can use their own Subscription manager OR@n
 *      Default Subscription manager provided by framework.
 */
// Instantiating custom Subscription Manager
Sec::Shp::Notification::ISubscriptionDB *subDBStore = NULL;
#ifdef USE_SQLITE3_SUBS_MANAGER
    subDBStore = Sec::Shp::Notification::SHPSqliteSubscriptionDB::createInstance();
#elif USE_FILE_SUBS_MANAGER
    subDBStore = Sec::Shp::Notification::SHPFileSubscriptionDB::createInstance();
#else
#error "implement a custom Subscription Manager and configure it with framework"
#endif

Sec::Shp::Notification::ISubscriptionManager *pSub =
Sec::Shp::Notification::SHPSubscriptionManager::createInstance(subDBStore);

/**
 * @note To Use Default Subscription Manager, please un-comment below line and comment above line
 */
if (false == sp_shp->setSubscriptionManager(*pSub)) {
    std::cout << "Failed to set Subscription Manager instance" << std::endl;
    return false;
}
```

```
/** Configure Subscription DB Path */
if (false == p_config->setSubscriptionDbPath("SubscriptionManager.db")) { // User needs to give
actual DB Path
        std::cout << "Failed to set Subscription DB Path" << std::endl;
        return false;
```

2.4.1.2.6. Setting of Remote Access Configuration File Path

`Sec::Shp::Configuration::setRAConfigPath()` is the API to be used for setting Remote Access Configuration File Path

This file is used to specify remote access configuration details (**if remote access feature is configured for the application**). By default, these files are stored in the *config* folder under "SDK Components directory path" preference field value. If the application developer wants to change the file path, update "RA_CONFIG_FILE_PATH" macro to the new file path.

```
#define RA_CONFIG_FILE_PATH
"/home/SHP_SDK/Lastest/SHP_Framework/Linux_32_Bit/config/controllable.ra.config"
/**< Represents Remote Access Configuration file path */
```

Note: It is application developers' responsibility to ensure existence (if required create before execution of the application itself) of the parent directory (for e.g., */home/SHP_SDK/configs/)* of '*remoteconfigs*' folder. Framework will only be able to create folders '*remoteconfigs*' onwards.

```
#ifdef REMOTE_ACCESS_SUPPORT
if (false == p_config->setRemoteAccessEnable(true)) {
        std::cout << "Failed to enable Remote Access Support" << std::endl;
        return false;
}
/** Set Remote Access Configuration File Path */
std::string raConfigFilePath = RA_CONFIG_FILE_PATH;

/**
 * Configure Remote Access Configuration File::@n
 *
 * 1) No encryption OR@n
 *    - 'raConfigFilePath' will not be encrypted when the second private key argument is not
provided@n
 */

if (false == p_config->setRAConfigPath(raConfigFilePath.c_str())) {
   std::cout << "Failed to set Remote Access Configuration path" << std::endl;
   return false;
}
```

2.4.1.2.7. Configuration of factories

`Sec::Shp::Configuration::setDeviceFactory(),Sec::Shp::Configuration::setSerializableDataFactory(), and Sec::Shp::Configuration::setResourceHandlerFactory()` are the APIs to be used for configuring factories

By default, the application sets the device, serializable, and resource handler factories. Application developers can use their own implementation by setting above mentioned APIs. . If application developer wants to configure custom factories then they are expected to implement respective interface (*Sec::Shp::DeviceFinder::DeviceFactory* for custom device factory) and modify following calls in *SHPUtils::initializeFramework()* method of file: **<*SHP-C++_Project_Name*>\SHPUtils.cpp**.

```

/** Configure Factories :: */
if (false == p_config->setDeviceFactory(new ::SHPDeviceFactory())) {
        std::cout << "Failed to set SHP Device Factory" << std::endl;
        return false;
}

if (false == p_config->setSerializableDataFactory(new ::SHPSerializationFactory())) {
        std::cout << "Failed to set SHP Serializable Data Factory" << std::endl;
        return false;
}

if (false == p_config->setResourceHandlerFactory(new ::SHPResourceHandlerFactory())) {
        std::cout << "Failed to set SHP Resource Handler Factory" << std::endl;
        return false;
}
```

2.4.1.2.8. Configuration of Server Connectors

`Sec::Shp::Connector::SSLConfiguration::addCACertificate()`, `Sec::Shp::Connector::SSLConfiguration::setSelfCertificateWithRSAPrivateKey()`, are the APIs to be used for configuring HTTPS server connectors **and** `Sec::Shp::Configuration::setServerConnector()` is the API to be used for setting configured server connector as the server connector of SHP Framework

By default, the application configures and sets server connectors based on the option (internal http/https, FCGX, or custom) selected while creation of the project. Application developers can implement their own custom server connector by implementing `Sec::Shp::Connector::Server::IServerConnector` and set using `setServerConnector()` API in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**.

```
/** Configure Client AND Server Connectors :: */
#ifdef  USE_INTERNAL_HTTPS_SERVER
    std::string serverRootCA              = SERVER_ROOT_CA;
    std::string serverCertificate         = SERVER_SELF_CERTIFICATE_RSA;
    std::string serverRSAPrivateKey       = SERVER_SELF_CERTIFICATE_PRIVATE_KEY_PATH;
    std::string serverRSAKeyFilePassword = SERVER_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD;

    /** Configure Server Certificates */
    Sec::Shp::Connector::Server::IServerConnector *pServerConnector =
Sec::Shp::Connector::Server::SHPHTTPSServerConnector::createInstance(deviceIPAddress, port);
    if (NULL == pServerConnector) {
        std::cout << "Failed to get Server Connector object" << std::endl;
        return false;
    }

    Sec::Shp::Connector::SSLConfiguration *pServerConnectorConfiguration =
(Sec::Shp::Connector::SSLConfiguration *) (pServerConnector->getConnectorConfiguration());
    if (NULL == pServerConnectorConfiguration) {
        std::cout << "Failed to get Client SSL Configuration object" << std::endl;
        return false;
    }

    if (false == pServerConnectorConfiguration->addCACertificate(serverRootCA.c_str())) {
        std::cout << "Failed to set Server Root CA Certification" << std::endl;
        return false;
    }

    if (false == pServerConnectorConfiguration-
>setSelfCertificateWithRSAPrivateKey(serverCertificate.c_str(), serverRSAPrivateKey.c_str(),
serverRSAKeyFilePassword.c_str())) {
        std::cout << "Failed to set Server Self Certificate/Key" << std::endl;
        return false;
    }

    if (false == p_config->setServerConnector(*pServerConnector)) {
        std::cout << "Failed to set Internal HTTPS Server Connector" << std::endl;
        return false;
    }
#elif  USE_FCGI_HTTP_CONNECTOR
    Sec::Shp::Connector::Server::IServerConnector *connector = new
Sec::Shp::Connector::Server::SHPFCGXServerConnector::createInstance(true);
    if (false == p_config->setServerConnector(*connector)) {
        std::cout << "Failed to set FCGI Server Connector" << std::endl;
        return false;
    }
#else
#error "implement a custom Server Connector and configure it with framework"
#endif
```

2.4.1.2.9. Configuration of Client Connectors

`Sec::Shp::Connector::SSLConfiguration::addCACertificate(),` `Sec::Shp::Connector::SSLConfiguration::setSelfCertificateWithRSAPrivateKey(),` are the APIs to be used for configuring HTTPS client connectors **and** `Sec::Shp::Configuration::setClientConnector()` is the API to be used for setting configured client connector as the client connector of SHP Framework

By default, the application configures and sets server connectors based on the option (internal http/https, or custom) selected while creation of the project. Application developers can implement their own custom server connector by implementing `Sec::Shp::Connector::Client::IClientConnector` and set using `setClientConnector()` API in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**.

```
#ifdef  USE_INTERNAL_HTTPS_CLIENT
    std::string clientRootCA              = CLIENT_ROOT_CA;
    std::string clientCertificate         = CLIENT_SELF_CERTIFICATE_RSA;
    std::string clientRSAPrivateKey       = CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_PATH;
    std::string clientRSAKeyFilePassword = CLIENT_SELF_CERTIFICATE_PRIVATE_KEY_FILE_PASSWORD;

    /** Configure Client Certificates */
    Sec::Shp::Connector::Client::IClientConnector*       pClientConnector =
Sec::Shp::Connector::Client::SHPHTTPSClientConnector::createInstance();
    Sec::Shp::Connector::SSLConfiguration *pClientConnectorConfiguration =
(Sec::Shp::Connector::SSLConfiguration *) (pClientConnector->getConnectorConfiguration());
    if (NULL == pClientConnectorConfiguration) {
        std::cout << "Failed to get Client SSL Configuration object" << std::endl;
        return false;
    }

    if (false == pClientConnectorConfiguration->addCACertificate(clientRootCA.c_str())) {
        std::cout << "Failed to set Client Root CA Certificate" << std::endl;
        return false;
    }
#ifdef REMOTE_ACCESS_SUPPORT
    std::string commonRootCA = COMMON_ROOT_CA;
    if (false == pClientConnectorConfiguration->addCACertificate(COMMON_ROOT_CA)) {
        std::cout << "Failed to set Common Root CA Certificate which need to access External
Cloud Servers" << std::endl;
        return false;
    }
#endif
    if (false ==  pClientConnectorConfiguration-
>setSelfCertificateWithRSAPrivateKey(clientCertificate.c_str(), clientRSAPrivateKey.c_str(),
clientRSAKeyFilePassword.c_str())) {
        std::cout << "Failed to set Client Self Certificate/Key" << std::endl;
        return false;
    }

    pClientConnectorConfiguration->enablePeerVerification();

    if (false == p_config->setClientConnector(*pClientConnector)) {
        std::cout << "Failed to set Internal HTTPS Client Connector." << std::endl;
        return false;
    }
#else
#error "implement a custom Client Connector and configure it with framework"
#endif
```

2.4.1.2.10. Verification and Setting of final Configuration

`Sec::Shp::Configuration::setConfiguration()` is the API to be used for setting final configuration with SHP Framework

```
sp_shp->setConfiguration(p_config);
```

2.4.2. Creation, initialization, and setting of SHP Framework listeners

SHP Framework makes use of configured listeners for notifying status events, SHP device related notifications. By default, the application will create, initialize, and set SHP Status Listener (file: ***<SHP-C++_Project_Name>*/SHPListener.cpp**). However, the application developers (especially controller application) are expected to implement device finder listener for handling all SHP device related notifications from SHP Framework.

2.4.2.1. Creation and configuration of Device Finder Listener

`Sec::Shp::SHP::getDeviceFinder()->setDeviceFinderListener()` is the API to be used for setting custom device finder listener with SHP Framework

By default, application will not generate code for handling device related notifications. However, developers can follow below mentioned guideline in *SHPUtils::initializeFramework()* method of file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp)**

Developers are expected to implement '*Sec::Shp::IDeviceFinderListener*' for getting and handling all SHP devices related notifications, like, new SHP device discovery, modification to existing SHP device, and SHP device leaving network (*add/modify/delete*). Controller applications are mostly expected to create, configure, and set their custom device finder listeners to discover peer SHP devices and do specific actions based on received notifications.

Following is the sample implementation of '*Sec::Shp::IDeviceFinderListener*' :

```
/**
 * Developers need to implement 'IDeviceFinderListener' for getting device related
(add/modify/delete)
 * notifications from SHP-Framework.
 * @n
 * If the application type is a Controller then handling device related notifications is mostly
 * expected.
 * @n
 * @note Un-comment following block of code for setting device finder listener (before that
developers are expected to implement 'IDeviceFinderListener'
 */

class DeviceFinderListener : public Sec::Shp::DeviceFinder::IDeviceFinderListener
{
        virtual void OnDeviceAdded( Sec::Shp::Device& device )
        {
                /* This method will be invoked when a new device is discovered */
        }
        virtual void OnDeviceRemoved( Sec::Shp::Device& device)
        {
                /*This method will be invoked when a device leaves the network */
        }
        virtual void OnDeviceUpdated(Sec::Shp::Device& device)
        {
                /*This method will be invoked when a device details got updated */
        }
};
```

Following is the sample registration of custom device finder listener with SHP Framework.:

```
sp_shp->getDeviceFinder()->setDeviceFinderListener( *(new DeviceFinderListener()) ) ;
```

2.4.2.2. Creation and configuration of SHP Status Listener

`Sec::Shp::SHP::setSHPListener()` is the API to be used for setting SHP Status Listener with SHP Framework

By default, the application will generate code for receiving and handling status related notifications from SHP Framework and providing configuration data back to SHP Framework. Applications will create, initialize, and set the SHP Status Listener (file: **<*SHP-C++_Project_Name*>\SHPListener.cpp**).

Developers can also have their own custom SHP Listener by implementing '*Sec::Shp::ISHPListener*' and handle SHP Framework notifications related to status (start/ stop / error) of framework, easy setup, registration, and device token related. Developers are expected to set their custom SHP Status listener by using above mentioned API `Sec::Shp::SHP::setSHPListener()`.

```
/**
 * Configure SHP Listener
 */
if (false == sp_shp->setSHPListener(*sp_shpListener)) {
   std::cout << "Failed to set SHP Listener Object" << std::endl;
   return false;
}
```

2.4.2.2.1. More details on 'ISHPListener' interface

The *ISHPListener* interface is used for receiving notification from the SHP Framework and providing configuration data to SHP Framework. As part of project creation, the *SHPListener* class is created which implements *ISHPListener*. *ISHPListener* interface has the following methods (APIs).

Note: Complete description and documentation is available with SHP Framework API documentation

public void onStarted(String address)

This method will be invoked by framework when framework is started. In return, it gets address of the server which is started by framework.

public void onStopped()

This method will be invoked when framework is stopped.

public void onError()

This method will be invoked when framework encounters an error.

public String getMyDeviceToken()

This method will be invoked by SHP Framework for getting Controlled devices token

public String getUUIDAndTokenMap()

This is no longer used.

public void updateUUIDAndTokenMap(Sec.Shp.DeviceToken token)

This method is used only on Controller Application. This method will be invoked whenever SHP Framework receive device token and pass received device token and UUID of controlled Device.

public void tokenRequestNotification(String uuid)

This method will be invoked whenever SHP Framework gets request for Device Token and pass UUID of the requested device.

public void easySetupNotification(Sec.Shp.EasySetupNotifactionnotif)

This method will be invoked by SHP Framework for notifying the status of easy setup and registration & de-registration procdure.

void scsErrorNotification(interroCode)

This method will be invoked by SHP Framework for notifying the error code which was sent by SCS server.

public void OnChannelCreated(String channelName, intconnID)

This method will be invoked whenever steam channel is created by SHP framework and provide Stream *ChannelName* and connection Identifier to application.

public String getAuthCode()

This method is used only on Controller Application. SHP Framework will invoke for getting “Authcode” from application.

public String getAccessToken()

This method is used only on Controller Application. SHP Framework will invoke for getting “AccessToken” from application.

public String getRefreshToken()

This method is used only on Controller Application. SHP Framework will invoke for getting “Refresh Token” from application.

public String getUserID()

This method is used only on Controller Application. SHP Framework will invoke for getting “UserID” from application.

Following is the code snippet of applications default *ISHPListener* implementation present in file: ***<SHP-C++_Project_Name>*\SHPListener.cpp**).

```
#ifndef __SHPListener__
#define __SHPListener__
#include <iostream>
#include "ISHPListener.h"
#include "ConditionUtility.h"
#include "MySHPDevice.h"
/**
 * @classSHPListener
 * @briefThis class implements ISHPListener, this class is used for receiving notifications
 *    from SHP Framework and providing configuration data to SHP Framework.
 */
class SHPListener: public Sec::Shp::ISHPListener
{
        public:
            /**
             * Default constructor of SHPListener
             */
        SHPListener();
            /**
             * Default destructor of SHPListener
             */
        virtual ~SHPListener();
        /**
         * This method will be a notified from SHP-Framework to application along with the UUID
         * from which device token request is received.
         * This method is used only on Controlled Application.
         *
         * @param[in]    uuidUUID  from which device token request is received by SHP-Framework
         */
        void tokenRequestNotification(std::string uuid);
```

2.4.3. Starting SHP Framework

`Sec::Shp::SHP::start()` is the API to be used for starting SHP Framework after successful initialization SHP Framework AND `Sec::Shp::SHP::getSHPState()` is the API to be used for knowing current execution status of SHP Framework

SHP Framework *start()* method is a non-blocking call and the framework will be in *Sec::Shp::SHP_STARTING* state till SHP is completely started. The SHP Framework will go to state *Sec::Shp::SHP_STARTED* once it is completely and successfully started all servers, clients, and successfully registered the device with the cloud. Developers are expected not to perform any other activity till SHP is completely started. By default, the application will be waiting for SHP to either to start successfully or exit with an error. `Sec::Shp::SHP::getSHPState()` needs to be used by developers to know status of SHP Framework any time.

Note: Complete documentation on valid SHP States are explained in detail in SHP API documentation

Once the SHP is successfully started, the application checks whether Easy Setup routine needs to be performed or not by checking the flag '*easySetupRequired*' of *SHPUtils* class. By default, before initialization of the framework, the application checks whether application (running on device) is provisioned or not. It will verify whether application possess all required details to be connected to Home Access Point (Home AP) for registering onto cloud.

- If application is not provisioned then enable Soft AP mode on device and perform Easy Setup routine (refer to SHP Architecture for complete details on Easy Setup routine)

SHPUtils::startFramework() method in file: ***<SHP-C++_Project_Name>*\SHPUtils.cpp**) does contain default implementation of application for invoking SHP Framework start and based on need initiates Easy Setup routine (*SHPUtils::performEasySetupProcess()*).

```
bool
SHPUtils::startFramework()
{
    if ((false == isFrameworkIntialized) || (NULL == sp_shp) || (NULL == sp_myDevice)) {
        cout<<"SHPUtils::startFramework() => "<<"ERROR: SHP Framework hasn't initialized"<<
std::endl;      return false;
    }
    if (false == sp_shp->start(*sp_myDevice)) {
        cout << "SHPUtils::startFramework() => "<<"ERROR: Failed to Start framework"<<std::endl;
        return false;
    }
    /** Confirm SHP-Start */
    Sec::Shp::SHPStates shpState = sp_shp->getSHPState() ;

    while (Sec::Shp::SHP_STARTING == shpState) {
        Sleep(1000);  // Waiting for 1 second for checking status of SHP
        cout << "SHPUtils::startFramework() => " << "INFO - Waiting for SHP to be started
completely, SHP Running Status " << shpState << std::endl;
        shpState = sp_shp->getSHPState() ;
    }

    if (shpState == Sec::Shp::SHP_STARTED) {
        std::cout << "SHPUtils::startFramework() => " << "INFO - Successfully started SHP with
State " << shpState << std::endl;
    }
    else {
        std::cout << "SHPUtils::startFramework() => " << "ERROR: Failed to Start framework with
State " << shpState << ", Hence Exiting!!!" << std::endl;
        stopFramework();
        return false;
    }
#ifdef REMOTE_ACCESS_SUPPORT
    if (easySetupRequired == true) {
        performEasySetupProcess();
    }
#endif
    return true;
}
```

Application sets '*easySetupRequired*' flag to true upon non-availability of Home AP details.

`SHPUtils::isWifiDetailsAvailable(),`

`SHPUtils::connectToHomeAccessPoint(),`

`SHPUtils::enableSoftAPMode(), and`

`SHPUtils::disableSoftAPMode()` are the application methods which needs to be implemented by application developers. What is expected out of each of these methods is clearly documented inline in the application project.

```
/**
 * Initially, application will check whether WiFi details are present or not to connect HomeAP.
 * If WiFi details are present, then device will connect to HomeAP. Application will terminate
 * on failure of connecting HomeAP. If device doesn't have WiFi information, then application
 * will exit in case of controller and device will goto softAP mode in case of controlled device.
 */
bool wifiDataAvailality = isWifiDetailsAvailable();
bool connected = false;

if (wifiDataAvailality == false) {
        if (p_config->getAppType() == Sec::Shp::ApplicationType_Controller) {
                cout << "SHPUtils::initializeFramework() => " << "No WiFi Details to Connect Home
                AP " << std::endl;
                return false;
        }
        else {
                cout << "SHPUtils::initializeFramework() => " << "No WiFi Details to Connect Home
                AP " << std::endl;
                if (false == enableSoftAPMode()) {
                        cout << "SHPUtils::initializeFramework() => " << "Failed to Start Soft-AP
                        Mode " << std::endl;
                        return false;
                }
                easySetupRequired = true;
                connected = true;
        }
}
else {
        connected = connectToHomeAccessPoint();
}

if (connected == false) {
        cout << "SHPUtils::initializeFramework() => " << "Failed to connect Home Access Point,
        Please check WiFi Details" << std::endl;
        return false;
}
```

2.4.4. Discovering devices

2.4.4.1. Retrieving discovered devices

The essential resources to be implemented for the device to be discovered by the SHP Plugin are *Capability* and *Devices*. The Capability Resource Handler is already implemented, so the application developer needs to implement the Devices Resource Handler. This can be done by implementing the required methods in the *DevicesResourceHandler.cpp* in the Server folder (*ShpGen* → *Server*). The status code must also be updated while implementing the Resource Handler methods.

Implement the following methods, updating status code and filling the '*respData*' structure.

For C++ Projects:

```
bool
DevicesResourceHandler::onGET( Sec::Shp::Connector::Server::ServerSession& session, int&
statusCode, ::Devices* respData)
{
        // TODO: Autogenerated code. Add Resource implementation here
        // TODO: Default Status Code is: 501 - Not Implemented.  Replace Default Status Code
        based on implementation!
        statusCode = 501;
        return true ;
}
```

For example, developers can make use of sample *device* implementation available in method *MySHPDevice::getDevices()*, for example:

```
bool
DevicesResourceHandler::onGET( Sec::Shp::Connector::Server::ServerSession& session, int&
statusCode, ::Devices* respData)
{
        std::cout<<"\n\n$$$$$$$$$$$$$$$$$$$  onGET Devices\n\n";
        statusCode = 200;
        return MySHPDevice::getInstance()->getDevices(respData);
}
```

Implement following methods, updating status code and storing/updating as per '*reqData*'.

For C++ Projects:

```
bool
DevicesResourceHandler::onPOST( Sec::Shp::Connector::Server::ServerSession& session, int&
statusCode, ::Device* reqData,std::string& location)
{
        // TODO: Autogenerated code. Add Resource implementation here
        // TODO: Default Status Code is: 501 - Not Implemented.  Replace Default Status Code
        based on implementation!
        statusCode = 501;
        return true ;
}
```

2.4.5. Performing Resource Control / Monitor / Manipulation

2.4.5.1. Sending REST requests to discovered devices

If application wants to send requests to any specific resource (*controlled applications*) of say discovered devices then

1. Expected to implement all required methods in respective *I<Resource_Name>ResourceResponseListener* located in 'Client' folder (*ShpGen* → *Client* in case of C++ Projects)

```
class LightResourceResponseListener : public ILightResourceResponseListener
{
        public:
        bool onGetLight(int& requestId, int status, ::Light* pRespData)
        {
                std::string power = pRespData->mpLightPower->value;
                // Check if light is powered on
                if (power.compare("on") == 0)
                {
                        int requestId;
                        Light *pLight = new Light();
                        pLight->mpLightPower = new OnType();
                        pLight->mpLightPower->value = "Off";
                        // Power off the light
                        pLightResource->putLight( requestId , *pLight);
                }

                return true;
        }

        bool onPutLight(int& requestId, int status)
        {
                if (status == 204)
                {
                  // Light powered off successfully
                }
        }
};
```

2. They are expected to use respective resource classes (*Sec::Shp::Client::Resource::<Resource_name>Resource*) to make requests, and

```
LightResource* pLightResource = NULL;
void powerOffLight()
{
        pLightResource = device->createResource(RT_LIGHT);
        if ( NULL != pLightResource )
        {
                int requestId;
                pLightResource->addResponseListener(*(new LightResourceResponseListener()));
                pLightResource->getLight(requestId);
        }
}
```

Application developers must have the knowledge of input and output data format SHP requests only then can they properly type cast the response data. Application needs to type cast the response data to corresponding data class generated by SHP-SDK. SHP-SDK also generates resource specific controllers and status listeners. It is advisable to use these resource specific controllers and interfaces to avoid type casting.

2.4.5.2. Precautions when assigning attribute values into the resource object

1. Due to nature of the SHP specification if any of attribute values are not belongs to Enumeration type SDK will not validate whether the contents of the value is appropriate or not, but SDK do the basic type validation. For example, "NotificationEventType" value should be one of the following: "Created" or "Notified" or "Deleted". Main difference when defining a type of attribute is, whether the certain attribute can be generalized or not. When you see the value of "progress" attribute located under the "Operation" resource, you'll find out it's defined as a "String20" and SDK will only checks length of the string is not exceed more than 20. Because, representation or supported values for this attribute can be very different among the devices and thereby SHP specification just provides a string container with one single constraints which is maximum length of the string.
2. SDK will removes from the string all leading and trailing white-space characters. Each leading and trailing trim operation stops when a non-whitespace character is encountered. For example, if the assigned attribute value is "   2014-10-31T18:30:00   ", the SDK internally converts it into "2014-10-31T18:30:00".

2.4.5.3. Handling REST requests from *Controller* devices

If the application wants to handle requests on specific resource (*controlled applications*) then application developers are expected to implement all required methods in the respective resource handler *<Resource_Name>ResourceHandler.cpp* in 'Server' folder (*ShpGen* → *Server* in case of C++). The status code shall also be updated while implementing respective Resource Handler methods.

By default, generated application handles '*Capability*' resource and respective resource handlers are available in *ShpGen* → *Server* → *CapabilityResourceHandler.cpp*

Please refer to sample implementation for in Retrieving discovered devices for handling GET request on '*Devices*' resource.

2.4.6. Stopping SHP Framework

`Sec::Shp::SHP::stop()` is the API to be used for stopping SHP Framework completely after successful start of SHP Framework

`Sec::Shp::SHP::stop(true)` is the API to be used for stopping SHP Framework internally after successful start of SHP Framework

`Sec::Shp::SHP::removeSHPListener()` is the API to be used for un-subscribing SHP Status listeners

SHP Framework *stop()* method is a blocking call and the framework will be in *Sec::Shp::SHP_STOPPING* state till SHP is completely stopped. SHP Framework will go to state *Sec::Shp::SHP_STOPPED* once it is completely and successfully stopped all servers, and clients. Developers will not to be able to perform any other activity till SHP is completely stopped once '*stop()*' method is invoked. By default, the application will be verifying completeness of SHP Framework stop by checking `Sec::Shp::SHP::getSHPState()` with *Sec::Shp::SHP_STOPPED*.

Note: Complete documentation on valid SHP States are explained in detail in SHP API documentation.

Once the SHP is successfully stopped by default application, unsubscribe to SHP Status notifications by invoking API, `Sec::Shp::SHP::removeSHPListener()` and do memory cleanup.

SHPUtils::stopFramework() method in file: **<*SHP-C++_Project_Name*>\SHPUtils.cpp**) does contain default implementation of application for invoking SHP Framework stop.

```
bool
SHPUtils::stopFramework()
{
        if (sp_shp == NULL) {
                return false;
        }


        sp_shp->stop();

        /** Confirm SHP-Stop */
        Sec::Shp::SHPStates shpState = sp_shp->getSHPState() ;
        if (Sec::Shp::SHP_STOPPED == shpState ) {
                std::cout << "SHPUtils::stopFramework() => " << "INFO - SHP Stopped Completely"
                << std::endl;
        }


        /** Un-subscribe SHP Listener */
        sp_shp->removeSHPListener(*sp_shpListener);


        if (sp_shpListener) { delete sp_shpListener; }


        /** Reset SHP Configuration */
        Sec::Shp::Configuration *config = sp_shp->getConfiguration();


        if (NULL != config) {
                config->reset();
        }


#ifdef REMOTE_ACCESS_SUPPORT
        if (sp_condition) { delete sp_condition; }
#endif

        if (sp_shp) { delete sp_shp; }

        std::cout << "SHPUtils::stopFramework() => " << "Exiting Successfully!" << std::endl;
        return true;
}
```

Note: ***SHP Framework provides another variant of stop() method stop(true) which will make SHP Framework to perform an internal stop****.*

For complete details please refer to SHP-API documentation.

2.4.7. Implementation of network resource handler

As part of an easy setup procedure, Controller Device will PUT network resource with the Wi-Fi AP access information to controlled the Device. The application developer shall implement *onPUT()* method of *NetworkResourceHandler* class and stores Wi-Fi AP access information in persistent storage for avoiding losing of AP access information during restart of the device.

2.4.8. Remote Access Configuration Files

This file is used to specify remote access configuration details. By default, this files (controller.ra.config, controllable.ra.config) are stored in *config* folder under "SDK Components directory path" preference field value. This file contains default value for all fields. The application developer needs to modify this file if required. The file contains the following remote access configuration details.

AUTH_ACC_SERVER_ADDR

This field specifies the Authentication Account server address.

API_ACC_SERVER_ADDR

This field specifies the API Account server address.

ACC_SERVER_PORT

This field specifies the Account Server port Number.

SERVICE_SERVER_ADDR

This field specifies the Service Server Address.

SERVICE_SERVER_PORT

This field specifies the Service Server Port Number.

REMOTE_SERVICE_PORT

This field specifies the P2P service port for the connection

REMOTE_SERVER_TYPE

This field specifies type of the SCS server to be connected. During the development period, it's strongly suggested to use a staging server to avoid any interference with a commercial smart home service being operated. (0: Production, 1: Staging)

SCS_LOG_LEVEL

This field specifies the SCS library log level.

SCS_LOG_PATH

This field specifies the SCS log folder, where SCS library create log file and write logs to file.

REMOTE_CONFIG_PATH

This field specifies the folder, where SHP Framework create file for storing SCS configuration details like authCode, accountID, peerID, peerGroupID, countryCode and guid(Global User ID).

For example, ***REMOTE_CONFIG_PATH=/home/SHP_SDK/configs/remoteconfigs/***

Note: It is application developers' responsibility to ensure existence (if required create before execution of the application itself) of the parent directory (for e.g., */home/SHP_SDK/configs/)* of '*remoteconfigs*' folder. Framework will only be able to create folders '*remoteconfigs*' onwards.

Developer Note:

AppID

This value specifies the application id. If application developer wants to get a new application id, please contact to the Convergence Development Group of the MSC division.

AppSecret

This value specifies the application secret key that depends on the application id. When application developer gets an application id, corresponding application secret key also will be provided.

InstanceID

This value specifies the application instance id that represents purpose of the application. If application developer wants to get a new instance id, please contact to the Convergence Development Group of the MSC division.

Above three values shall be set from application using their respective setter APIs, for reference, `RemoteAccessConfig::setAppId()` , `RemoteAccessConfig::setAppSecret()`, AND `RemoteAccessConfig::setInstanceId()`.

2.4.9. Setting of Authorization Grant Type

Please note that user must create a Samsung Account to utilize a remote access feature. Please visit "https://account.samsung.com/membership/signUp.do" to sign up to Samsung Account.

Application should set required authorization grant type by using the API below:

C++ API: **void RemoteAccessConfig::setAuthorizationType(const AuthorizationType authType)**

The application shall provide all mandatory details as part of the SHP framework initialization based on Authorization Grant Type. If any mandatory details are missing, framework throw error and stop framework. Please find mandatory parameters details for each Authorization Grant Type in table. And Instance ID with zero is not allowed by framework.

Authorization Grant Type	**Mandatory Details From App**	**C++ API for setting required details**
Resource Owner Password Credentials (AUTH_GRANT_TYPE_PASSWORD_CRED)	AppId , AppSecret, InstanceId, Email & Password. Framework will retrieve authcode, accessToken, RefreshToken & UserID by using provided credentials. Whenever current access token is expired, framework retrieves new access token by using refresh token.	RemoteAccessConfig::setEmail() for setting email id. RemoteAccessConfig::setPassword() for setting password. RemoteAccessConfig::setAppId() for setting App ID. RemoteAccessConfig::setAppSecret() for setting App Secret. RemoteAccessConfig::setInstanceId() for setting Instance ID.
Authorization Code (AUTH_GRANT_TYPE_AUTH)	AppId , AppSecret, InstanceId, Email, [AuthCode]. Framework will retrieve accessToken, RefreshToken & UserID by using provided credentials. If application does not set “AuthCode” as part of initialization, framework will call ISHPListener::getAuthCode() callback for getting “AuthCode” from application. Whenever current access token is expired, framework retrieves new access token by using refresh token.	

Implicit Grant (AUTH_GRANT_TYPE_IMPLICIT)	AppId , AppSecret, InstanceId, Email, [AccessToken] & [UserID]. If application does not set "AccessToken" or "UserID" as part of initialization, framework will call ISHPListener:: getAccessToken() callback for getting "AccessToken" and ISHPListener::getUserID() for "UserID" from application. Whenever current access token is expired, framework will call ISHPListener::getAccessToken () callback for getting "AccessToken" and ISHPListener::getUserID() for "UserID" from application.	RemoteAccessConfig::setAuthCode() for setting Auth Code. RemoteAccessConfig::setAccessToken() for setting Access Token. RemoteAccessConfig::setRefreshToken() for setting Refresh Token. RemoteAccessConfig::setUserID() for setting User ID.
Access Token & Refresh Token (AUTH_GRANT_TYPE_EXTENDED_TOKEN)	AppId , AppSecret, InstanceId, Email, [AccessToken], [RefreshToken] & [UserID]. If application does not set "AccessToken", RefreshToken or "UserID" as part of initialization, framework will call ISHPListener:: getAccessToken() callback for getting "AccessToken", ISHPListener::getRefreshToken() callback for getting "RefreshToken" and ISHPListener::getUserID() for "UserID" from application. Whenever current access token is expired, framework retrieves new access token by using refresh token.	RemoteAccessConfig::getAuthCode() for getting Auth Code. RemoteAccessConfig:: getUserID() for getting User ID. RemoteAccessConfig::getAccessToken() for getting Access Token. RemoteAccessConfig::getRefreshToken() for getting refresh token.

2.4.10. Easy Setup, Registration and Remote Access

Smart Home devices shall be connected to the Smart Home Network (Cloud Server) for providing Smart Home services. All Smart Home Devices needs to be registered with Cloud Server beforehand, EITHER for accessing / monitoring / controlling (*Controller* devices) other SHP devices information OR for being accessible / to be controlled (*Controlled* device) by other SHP devices.

All devices needs to connect to Home Access Point ('Home AP' – which is expected provide external connectivity) firstly and then register with the Cloud Server. But generally most of Home appliances just have limited user interface like display and/or user input method comparing to mobile phone and PC, so that it would be not easy to type password for connection to the home network.

SHP Specification provides **'Easy Setup'** procedure for Smart Home Appliances which has limited user interface to connect to the Smart Home network with help of a mobile device like a smart phone.

'Easy Setup' process is required for those kind of devices which cannot connect to 'Home AP' on their own OR for those devices which are not having provision (Display GUI) to key in 'Home AP' details by users. For example, most of the *Controlled* devices (home appliances) which are not having display need **'Easy Setup'** process. Since the *Controller* device (smart phone or tablet) can connect to the 'Home AP' by itself, **'Easy Setup'** process procedure is not needed beforehand. Registration process of the *Controller* device with the Cloud Server is almost same as the *Controlled* device registration except that the *Controller* device can perform registration process by itself without help of other devices.

Devices which can connect to 'Home AP' by themselves (*Controller* devices) can also help devices which cannot connect on their own (*Controlled* devices). **'Easy Setup'** process does facilitate the same, in this process *Controller* device helps *Controlled* devices provisioning and eventually their registration with Cloud Server.

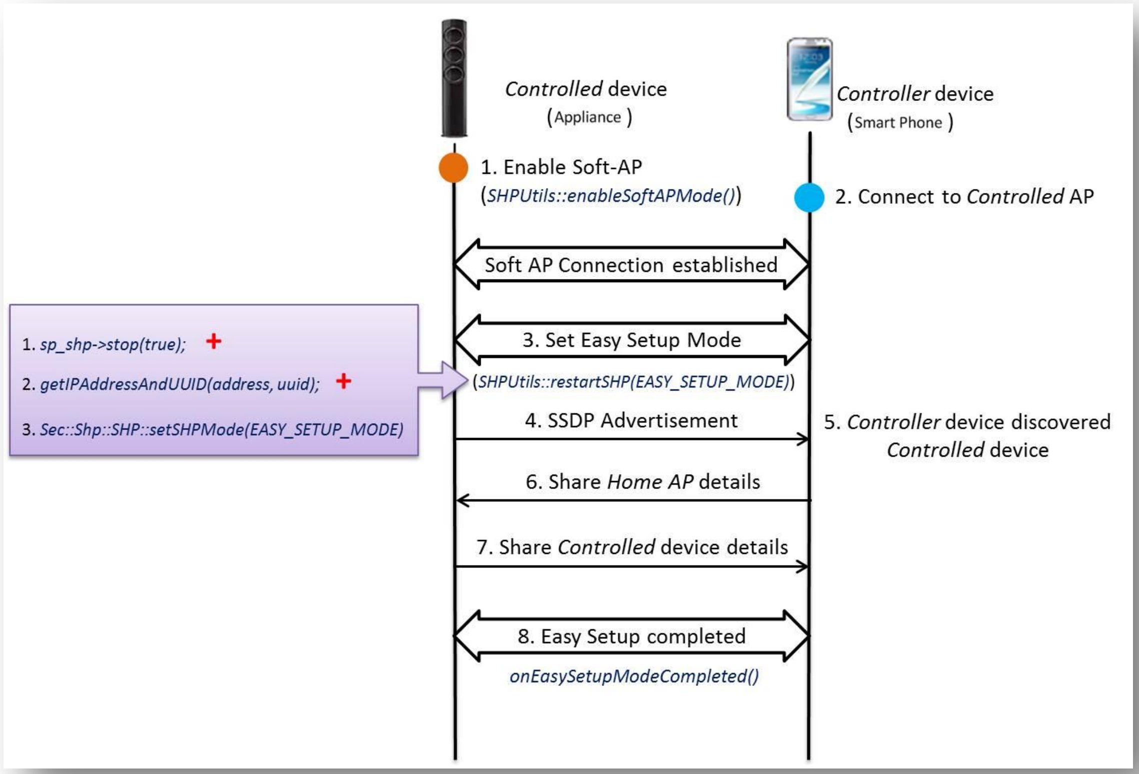
'Easy Setup' process mainly comprises of three phases:

1. **‘Easy Setup’**,
2. **‘Registration’**, and
3. ‘**SCS Login**’

2.4.10.1. Easy Setup Phase

In **‘Easy Setup’** phase,

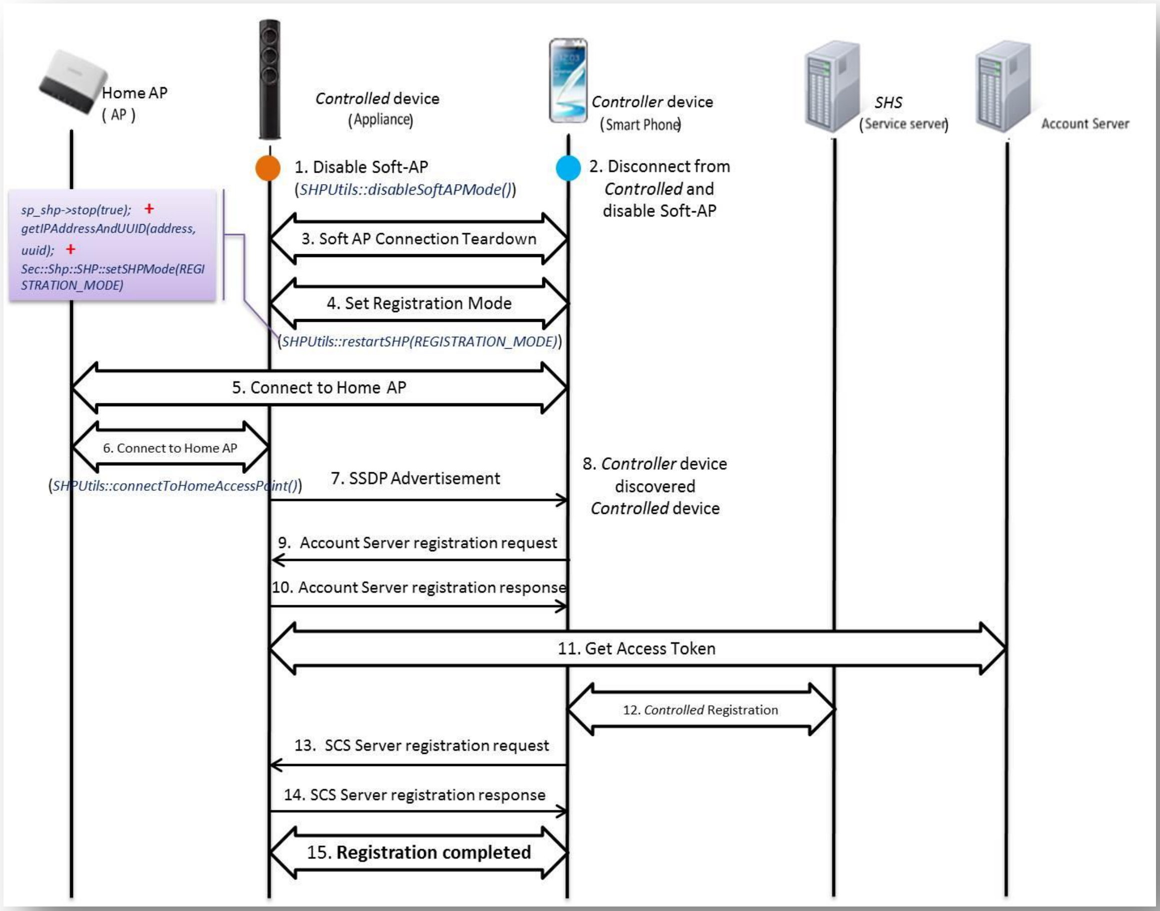
- ❖ Firstly, *Controller* (helper) device shall pair with the *Controlled* (device which needs external help for registration) device, SHP Specification supports two types of pairing:
    - ❖ **Soft-AP** connection mode, in which *Controlled* device acts as an AP and *Controller* device connects to *Controlled* device
    - ❖ **Wi-Fi P2P** connection mode, in which *Controller* device and the *Controlled* device make a direct connection with each other
- ❖ *Controller* device shall discover *Controlled* device
- ❖ Once discovered, *Controller* device provisions *Controlled* device by sharing details of ‘Home AP’
- ❖ And in return, *Controlled* device sends its *device* information to *Controller* for registration with Smart Home Server (SHS)
- ❖ Both *Controller* and *Controlled* device application receives ***onEasySetupModeCompleted()*** call back – this marks completion of **‘Easy Setup’** phase


Controlled device
(Appliance)
Controller device
(Smart Phone)
1. Enable Soft-AP
(SHPUtils::enableSoftAPMode())
2. Connect to Controlled AP
Soft AP Connection established
3. Set Easy Setup Mode
(SHPUtils::restartSHP(EASY_SETUP_MODE))
1. sp_shp->stop(true); +
2. getIPAddressAndUUID(address, uuid); +
3. Sec::Shp::SHP::setSHPMode(EASY_SETUP_MODE)
4. SSDP Advertisement
5. Controller device discovered Controlled device
6. Share Home AP details
7. Share Controlled device details
8. Easy Setup completed
onEasySetupModeCompleted()

2.4.10.2. Registration Phase

In **'Registration'** phase,

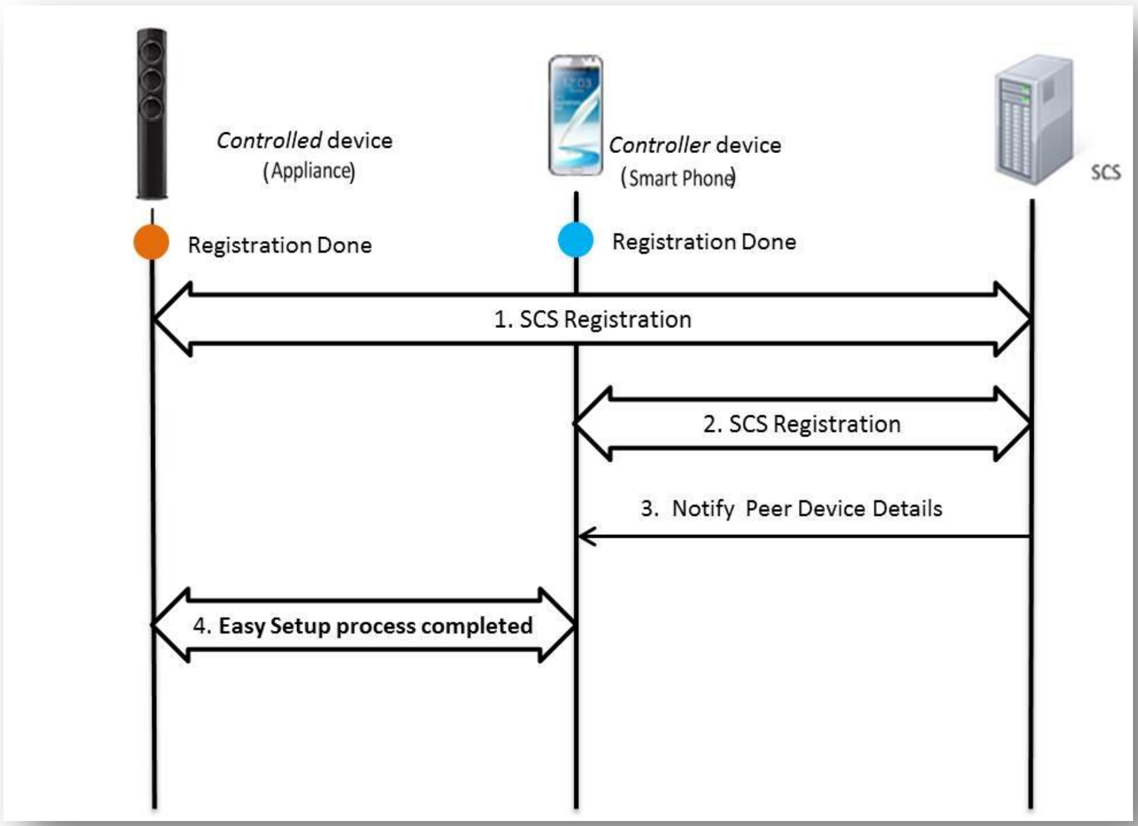
- Firstly, upon reception of ***onEasySetupModeCompleted()*** call back, both (*Controller* and *Controlled*) devices shall teardown their connection in Soft-AP mode (application needs to implement `SHPUtils::disableSoftAPMode()`) and connect back to the Home AP. Later, they are expected to initiate '**Registration**' phase by setting SHP-Framework mode to REGISTRATION_MODE
- *Controller* device shall discover *Controlled* device
- From Samsung Account server, *Controller* (helper) gets necessary credentials and sends (for example, *Authorization Code* and *E-mail ID*) to *Controlled* device – this will initiate Account server registration request on *Controlled* device
- Then *Controlled* device performs Account server registration and shares the response with *Controller*
- Upon the response, *Controller* performs *Controlled* device registration with Smart Home Server (SHS) by using the device details which are already retrieved in **'Easy Setup'** phase
- Upon successful registration, SHS server will return an ID (*peerID*) to the *Controller*
- Then, *Controller* device sends all necessary details (*peerID*, *peerGroupID*, *countryCode* and *etc.*) required for *Controlled* device to perform login with Samsung Smart Connectivity Server (SCS) – this will initiate SCS server registration request on *Controlled* device – this marks completion of **'Registration'** phase


Home AP
(AP)
Controlled device
(Appliance)
Controller device
(Smart Phone)
SHS
(Service server)
Account Server
1. Disable Soft-AP
(SHPUtils::disableSoftAPMode())
2. Disconnect from Controlled and disable Soft-AP
sp_shp->stop(true); + getIPAddressAndUUID(address, uuid); + Sec::Shp::SHP::setSHPMode(REGISTRATION_MODE)
3. Soft AP Connection Teardown
4. Set Registration Mode
(SHPUtils::restartSHP(REGISTRATION_MODE))
5. Connect to Home AP
6. Connect to Home AP
(SHPUtils::connectToHomeAccessPoint())
7. SSDP Advertisement
8. Controller device discovered Controlled device
9. Account Server registration request
10. Account Server registration response
11. Get Access Token
12. Controlled Registration
13. SCS Server registration request
14. SCS Server registration response
15. Registration completed

2.4.10.3. SCS Login Phase

In **'SCS Login'** phase,

- *Controlled* device performs SCS server registration using collected details in above two phases. *Controlled* device attempts SCS Login and update the same to *Controller* device, this marks completion of Easy Setup process. *Controlled* device makes use of these details for further initialization as well



2.4.10.4. Remote Access

In ‘**Remote Access’**,

- ❖ Upon successful SCS login, using ‘**Remote Access’** feature *Controller* device can manage (monitor/control) all registered *Controlled* devices through remote channel

Note: For complete details and usage of Remote Access feature, refer to Section 3.4.18

2.4.10.5. Easy Setup Notifications

As mentioned earlier during the **Easy Setup** process, SHP-Framework notifies each progress to the respective application (assumed to be having a listener by implementing *Sec:Shp::SHP::ISHPListener*) at each phase. This enables application developers to have their custom implementation (or actions to be done) based on the notification received. For example, upon reception of **EASY_SETUP_REQUIRED**, application developers are expected to initiate **Easy Setup** up process by enabling the device into Soft-AP mode and etc.

Note: Complete details about Easy Setup notifications are detailed in SHP-API documentation

Following are the possible notifications from SHP-Framework during **Easy Setup** process along with their explanation:

Easy Setup Notification	Description
DEVICE_TOKEN_REQUEST_TIMEOUT	Represents Device Token Request Timeout
EASY_SETUP_REQUIRED	Represents Easy setup is required.
DIFFERENT_COUNTRY_CODE	Represents Different Country Code.
AUTH_CODE_EXPIRED	Represents Authentication code is expired.
MISSING_MANDATORY_PARAMS	Represents Missing Mandatory Parameters.
LOCAL_SERVER_ERROR	Represents Local server ERROR.
NO_AUTHORIZATION_DETAILS	Represents No Authorization details.
CONNECTION_ERROR	Represents Connection ERROR.
REMOTE_SERVER_ERROR	Represents Remote Server ERROR.
INVALID_REQUEST	Represents Invalid Request.
CONNECTION_TIMEOUT	Represents Connection Time Out.
REFRESH_TOKEN_EXPIRED	Represents Refresh Token expired.
DEVICE_LOGIN_FAILED_TO_SCS	Represents Device Login failed to SCS.
FAILED_TO_START_TIMER	Represents Failed to start Timer.
FAILED_TO_REGISTER_DEVICE_TO_SHS	Represents Failed to register device to SHS.
FAILED_TO_FETCH_INFORMATION_FROM_SHS_CR	Represents Failed to fetch information from SHS.
FAILED_TO_UPDATE_REFRESH_TOKEN	Represents Failed to update refresh token.
ACCESS_TOKEN_EXPIRED	Represents Access token expired.

FAILED_TO_GET_TOKEN_DETAILS	Represents Failed to get Token details.
FAILED_TO_GET_AUTH_CODE_CR	Represents Failed to get Authentication Code.
EASY_SETUP_TIME_OUT_CR	Represents Easy setup Time Out.
DEVICE_NOT_REGISTERED	Represents Device not registered.
WAITING_FOR_HELPING_DEVICE_TO_BE_DISCOVERED_CR	Represents Waiting for helping device to be discovered.
AWAITING_WIFI_DETAILS_CD	Represents Awaiting WIFI details.
SENDING_WIFI_DETAILS_CR	Represents sending WIFI details.
WIFI_DETAILS_SENT_CR	Represents WIFI details sent.
REQUESTING_CONTROLLER_UUID_CD	Represents Requesting controller UUID.
SENDING_UUID_CR	Represents Sending UUID.
AWAITING_DEVICE_DETAILS_CR	Represents Awaiting device details.
SENDING_DEVICE_DETAILS_CD	Represents Sending device details.
EASY_SETUP_COMPLETED	Represents Easy setup completed.
REQUESTING_DEVICE_TOKEN_CR	Represents Requesting device token.
REGISTRATION_INITIATED	Represents Registration initiated.
GETTING_AUTH_CODE_FROM_SERVER_CR	Represents Getting Authentication code from server.
AWAITING_AUTHCODE_DETAILS_CD	Represents Awaiting authentication details.
SENDING_AUTHCODE_DETAILS_CR	Represents Sending authentication details.
GETTING_TOKEN_DETAILS_FROM_SERVER	Represents Getting token details from server.

GETTING_DEVICE_LIST_FROM_SHS_CR	Represents Getting device list from SHS.
GETTING_USER_INFO_FROM_SHS_CR	Represents Getting User info from SHS.
GETTING_PEERID_LIST_FROM_SHS_CR	Represents Getting peerID list from SHS.
DEVICE_REGISTERED_TO_SHS_CR	Represents Device registered to SHS.
WAITING_FOR_PEERID_DETAILS_CD	Represents Waiting for peer ID details.
SENDING_PEERID_DETAILS_CR	Represents Sending peer ID details.
ATTEMPTING_REGISTRATION_WITH_SCS	Represents Attempting registration with SCS.
DEVICE_LOGGED_INTO_SCS	Represents Device logged into SCS.
ALREADY_REGISTERED_TO_SHS_CR	Represents Already registered to SHS.
RETRYING_REGISTRATION_WITH_SCS	Represents Retrying registration with SCS.

2.4.10.6. Steps for Easy setup and Registration of *Controlled* Device

The SHP Framework should be started only when *Controlled* device's interface has proper IP Address. The application developer should follow steps for Easy setup and Registration.

1. Stop SHP Framework by calling *stop (true)* method of SHP class in case of SHP Framework is already started. As part of this method SHP-Framework stops all its Server Connectors and Remote Connectors.

2. Soft AP operation shall be started in the Controlled device whenever user pushes Soft AP enabling button.

3. Set SHP mode to Easy Setup Mode by calling *Sec::Shp::SHP::setSHPMode()* method of SHP class. The first argument value should be "**EASY_SETUP_MODE**" and second argument value should be IP address of Interface.

`SHPUtils::restartSHP(Sec::Shp::SHPModes mode)` is the method in file *<Project_Name>/SHPUtils.cpp* which performs internal stop [*stop(true)*] and takes *Sec::Shp::SHPMode* as parameters which will be passed as an argument to *Sec::Shp::SHP::setSHPMode()* method. Same method can be used for setting Easy Setup and registration routines. C++ application developers are expected to make use of this.

`SHPUtils::enableSoftAPMode()` is the method in file *<Project_Name>/SHPUtils.cpp* which needs to be implemented for enabling Soft AP during Step 2

4. Once receive "**EASY_SETUP_COMPLETED**" *easySetupNotification* through registered SHP listener *easySetupNotification()* method, stop SHP Framework by calling *stop (true)* method of SHP class.

`SHPUtils::processNotification()` is the method in file *<Project_Name>/SHPUtils.cpp* which handles all required easy setup notifications from SHP Framework. This method takes necessary action based on notification received, for example this method initiates registration process after receiving EASY_SETUP_COMPLETED notification from SHP Framework in Step 4

5. De-activate Soft AP operation and connect to Home AP using AP access information received from controlled Device.

`SHPUtils::disableSoftAPMode()` is the method in file *<Project_Name>/SHPUtils.cpp* which needs to be implemented for disabling Soft AP during Step 5

6. Set SHP mode to Registration Mode by calling *Sec::Shp::SHP::setSHPMode()* method of SHP class. The first argument value should be "**REGISTRATION_MODE**" and second argument value should be IP address of Interface. As part of registration procedure, SHP Framework will invoke "*getMyDeviceToken*()" method of registered SHP listener for getting Device Token, so application developer should implement "*getMyDeviceToken*()" method.

2.4.10.6.1. Function call constraints

It is strictly prohibited for application developers to call any framework API (especially *start()* and *stop()* methods) from framework callbacks to the application. They are requested to make use of their Application thread to perform any sort action OR write any implementation (GUI logic) which is time consuming.

Note: Doing some action on framework callback thread make SHP-Framework to be blocked till the finish of custom implementation.

For example, as aforementioned, during **Easy Setup** process and as part of few notifications, application is expected to restart SHP-Framework. However, application should not restart SHP framework from the callback thread (notification receiver).

One way of handling this by application developers is by setting some variable or state or by notifying application threads about the framework callback and relieve call back thread. Later, have implementation which does some action based on the variable/state/notification received on application thread only.

SHP-SDK handles this by making use of Conditional wait logic.

All generated C++ Controlled applications (generated using SHP-SDK-Plugin) will have a generated class *'ConditionUtility'* which implements functions related to Conditional wait logic. This is kind of conditional variable wrapper class. '*SHPUtility'* class of the generated application has *'ConditionUtility'* as a member, which means main thread will wait for the condition.

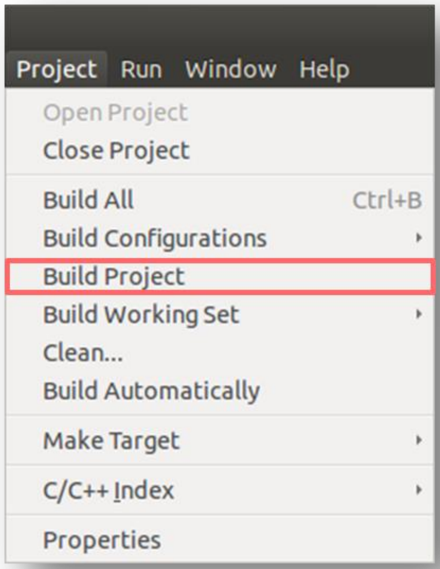
For example, upon reception of any Easy Setup notification (say, **EASY_SETUP_COMPLETED,** which expects application to restart framework) from framework it just notifies (*condition*) the application and relieves framework callback thread. And application will handle (restarts SHP-Framework by calling *sp_shp->stop(true)*) the received notification in its own thread.

```
class SHPUtils
{
public:
.
.
static ConditionUtility *sp_condition; /**< Represents object of ConditionUtility class.
*/
};

void
SHPListener::easySetupNotification(int eNotification)
{
    std::cout << "SHPListener::easySetupNotification(): => " << "Received Easy Set-up
Notification from Framework " << eNotification << std::endl;

    if (m_shpNotification != eNotification) {
        std::cout << "SHPListener::easySetupNotification(): => " << m_shpNotification <<
" : " << eNotification << std::endl;
        m_shpNotification = eNotification;
        std::cout << "SHPListener::easySetupNotification() => " << "Notifying Condition"
<< std::endl;
        SHPUtils::sp_condition->notify();
    }

    /**
     * TODO:    Application developers are expected to implement logic to handle
notifications from SHP-Framework and take necessary action.
     * @n
     * @note    By default code generator doesn't send anything
     */
}

bool
SHPUtils::processNotification()
{
#ifdef REMOTE_ACCESS_SUPPORT
    std::cout << "SHPUtils::processNotification() => " << "Called with Notification " <<
sp_shpListener->m_shpNotification << std::endl;

    /**
     * TODO: In this method, we are handling only easy setup process notification, if
application developer wants to
     * cover other notification, then he has to implement those.
     */
    if ((sp_shpListener->m_shpNotification == (int)Sec::Shp::NO_AUTHORIZATION_DETAILS) ||
            (sp_shpListener->m_shpNotification == (int)Sec::Shp::EASY_SETUP_REQUIRED))
    {
        std::cout << "SHPUtils::processNotification() => " << "No Authorization Details,
hence Device needs Easy Setup process..."
                    << "Do you want to start Easy Setup Process (Y/N)? : " << std::endl;
.
.
        restartSHP(Sec::Shp::EASY_SETUP_MODE);
    }
    else if (sp_shpListener->m_shpNotification == (int)Sec::Shp::EASY_SETUP_COMPLETED)
    {
        if (easySetupRequired == true) {
.
.
        restartSHP(Sec::Shp::EASY_SETUP_MODE);
    }

#endif
    return true;
}
```

2.5. Build and Run SHP Project

Before Building the project it needs to be ensured that all necessary library files are copied into the Project workspace

- SHP Eclipse plug-in does not provide any build & run tools. It invokes the build tools which are already installed in Eclipse
- After finishing the development build the project using Build menu command (Project → Build Project)



2.5.1. Execute the application

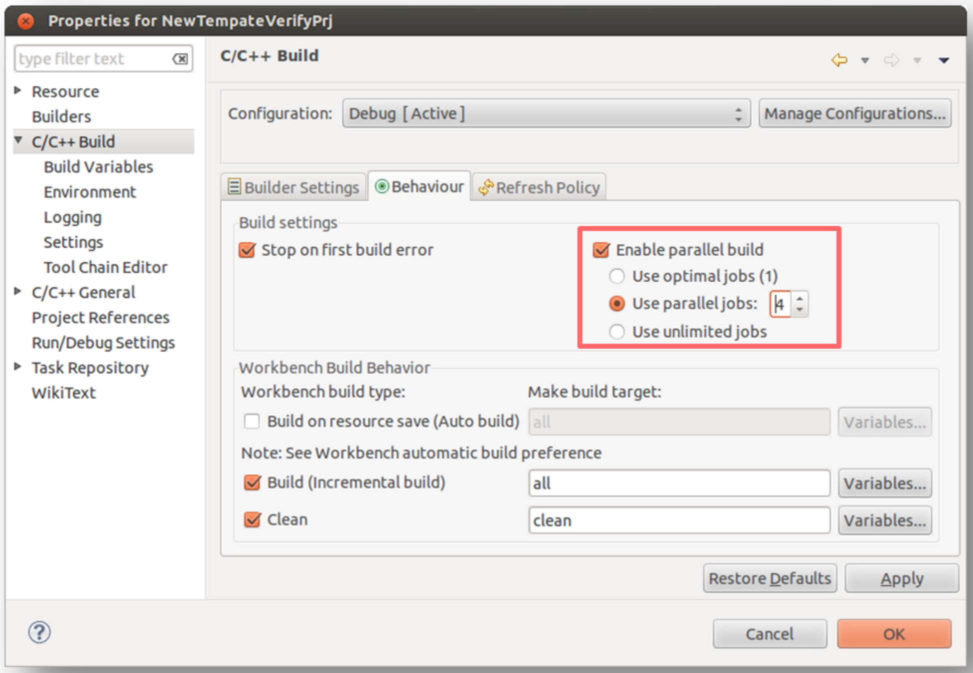
Unless user modify the default created functions *processNotification()* and *performEasySetupProcess()* in the SHPUtils.cpp, in case the application requires an Easy Setup, application must be executed on the shell to get the user input (Yes or No). Please set the LD_LIBRARY_PATH environment variable to corresponding libraries folder before executing the application.

For example, setting LD_LIBRARY_PATH environment variable in Linux 32 Bit Platform

```
$export LD_LIBRARY_PATH=<SHP_SDK_PATH>/SHP_Framework/Linux_32_Bit/sdk/cpp/lib
```

2.5.2. Fast Compilation in Eclipse

For fast compilation – 'build parallel compilation' option should be enabled in the eclipse build configurations.


Properties for NewTempateVerifyPrj

C/C++ Build

Configuration: Debug [Active]

Build settings
- [✓] Stop on first build error
- [✓] Enable parallel build
  - ( ) Use optimal jobs (1)
  - (●) Use parallel jobs: 4
  - ( ) Use unlimited jobs

Workbench Build Behavior

Workbench build type: / Make build target:
- [ ] Build on resource save (Auto build): all
- Note: See Workbench automatic build preference
- [✓] Build (Incremental build): all
- [✓] Clean: clean

2.6. Sample Application

The provided sample application show developers shows how to use The Smart Home SDK. For running the sample application, developers are expected to ensure existence/availability of the following on their development environment:

- Eclipse installation
- SHP-Plugin installation
  - Availability of SHP-libraries
- Required Tools installation
  - Compiler – G++/GCC
  - SQLite3 installation

Installation guides are in Chapter 1.

2.6.1. Features

Sample application has following additional features.

- How to implement a state machine with database. Please see the "DeviceManager.cpp" and files under the "/ShpGen/Database" directory to find out ways to manipulate database.

- How to handle REST calls. Additional code is implemented to the following files.

"/ShpGen/Server/DeviceResourceHandler.cpp"

"/ShpGen/Server/DevicesResourceHandler.cpp"

"/ShpGen/Server/InformationResourceHandler.cpp"

"/ShpGen/Server/OperationResourceHandler.cpp"

"/ShpGen/Server/TemperatureResourceHandler.cpp"

"/ShpGen/Server/TemperaturesResourceHandler.cpp"

"/ShpGen/Server/VersionResourceHandler.cpp"

"/ShpGen/Server/VersionsResourceHandler.cpp"

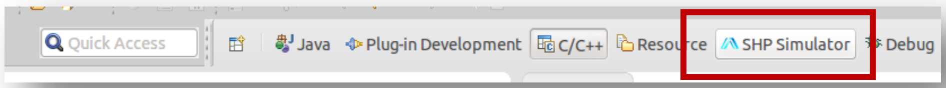
2.6.2. Build and run sample application

After importing the sample application project into the eclipse, it can be executed. Refer to 'Import existing SHP Projects into the Eclipse workspace' and 'Build and Run SHP Project' chapters.

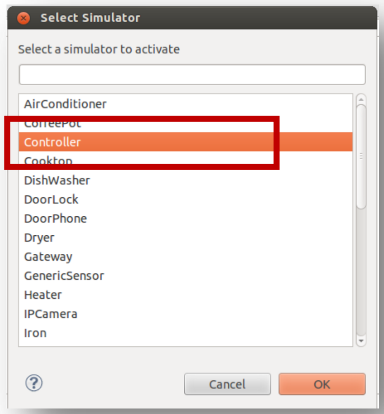
2.6.3. Testing sample application with Simulator

SHP Eclipse plug-in provides simulators which simulates the behavior of real SHP devices. Developer can use these simulators to test their application. Controlled application developers can use Controller Simulator for testing.

- Use "SHP Simulator perspective" to manage simulators. Please refer to section – 'Launching SHP Test Tool' and launch **"SHP Simulator perspective"**



- The provided sample application is '*Controlled*' application, so launch '*Controller*' Simulator, Please refer to section – 'Controller Test Tool' and select '*Controller*' Simulator, for example:


Select Simulator

Select a simulator to activate

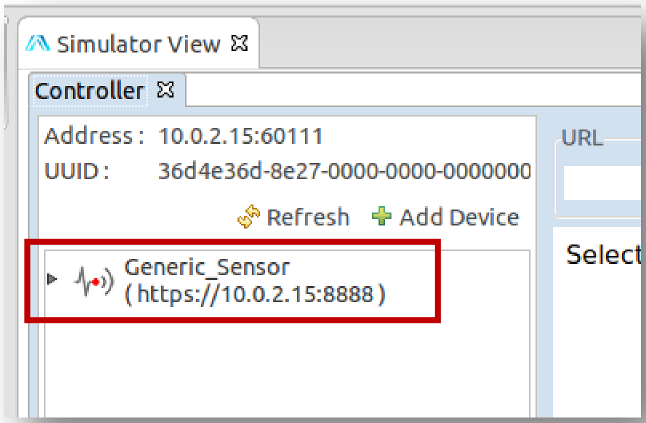
AirConditioner
CoffeePot
Controller
Cooktop
DishWasher
DoorLock
DoorPhone
Dryer
Gateway
GenericSensor
Heater
IPCamera
Iron

Cancel OK

NOTE 1: **Assuming that the provided sample application is '*Controlled-Generic Sensor*' application and rest of the screen shots describes further steps involved in testing sample SHP '*Controlled*' application using '*Controller*' simulators.**

NOTE 2: It is assumed that the sample **application is launched and running**

- Upon launching '*Controller*' simulator successfully, it shall discover and show developed '*Controlled*' application in the following way:


Simulator View

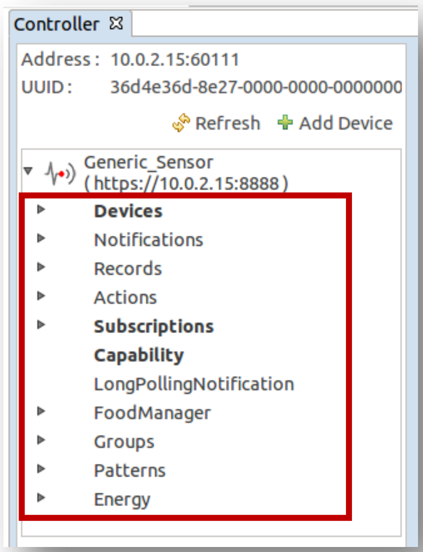
Controller

Address : 10.0.2.15:60111
UUID : 36d4e36d-8e27-0000-0000-0000000

Generic_Sensor
(https://10.0.2.15:8888)

NOTE: Developers can verify whether IP Address and port of the developed application (device) are shown correctly on '*Controller*' simulator or not

- Supported resources by the discovered device can be viewed on '*Controller*' simulator by expanding (clicking) on the device tree, it shall show resources of discovered device as follows:



Controller

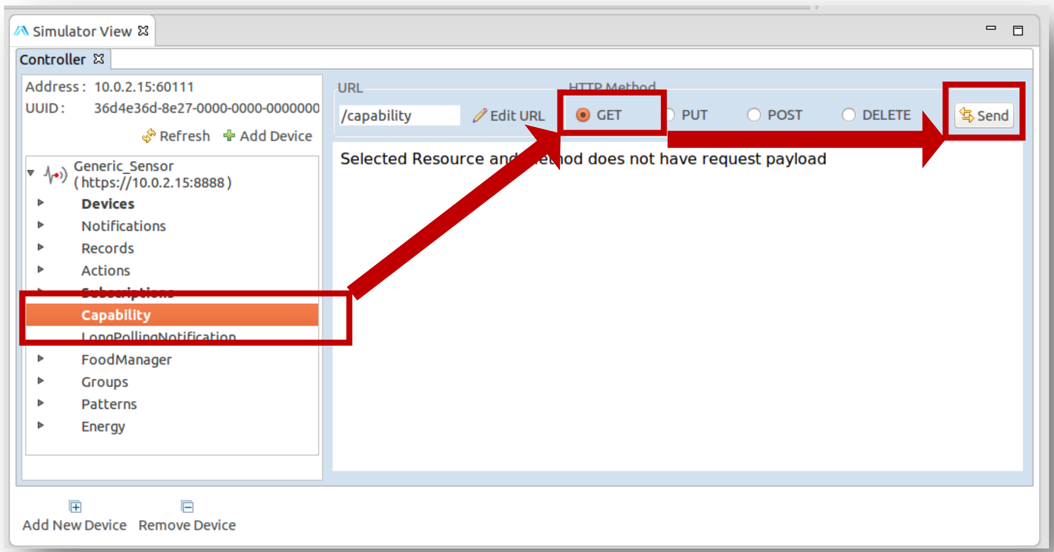
Address : 10.0.2.15:60111
UUID : 36d4e36d-8e27-0000-0000-0000000

Generic_Sensor (https://10.0.2.15:8888)

- Devices
- Notifications
- Records
- Actions
- Subscriptions
- Capability
- LongPollingNotification
- FoodManager
- Groups
- Patterns
- Energy

NOTE: Developers can verify whether the displayed resources are indeed supported by the developed application (device) on '*Controller*' simulator are correct or not

- Access/control developed application (device) from '*Controller*' simulator:
    - Select any one of the supported resources and try to get retrieve details, for example:
        1. Select 'Capability' resource
        2. Select GET request on 'HTTP Method'
        3. Click on 'Send' button



Simulator View

Controller

Address : 10.0.2.15:60111
UUID : 36d4e36d-8e27-0000-0000-0000000

Generic_Sensor (https://10.0.2.15:8888)

- Devices
- Notifications
- Records
- Actions
- Subscriptions
- Capability
- LongPollingNotification
- FoodManager
- Groups
- Patterns
- Energy

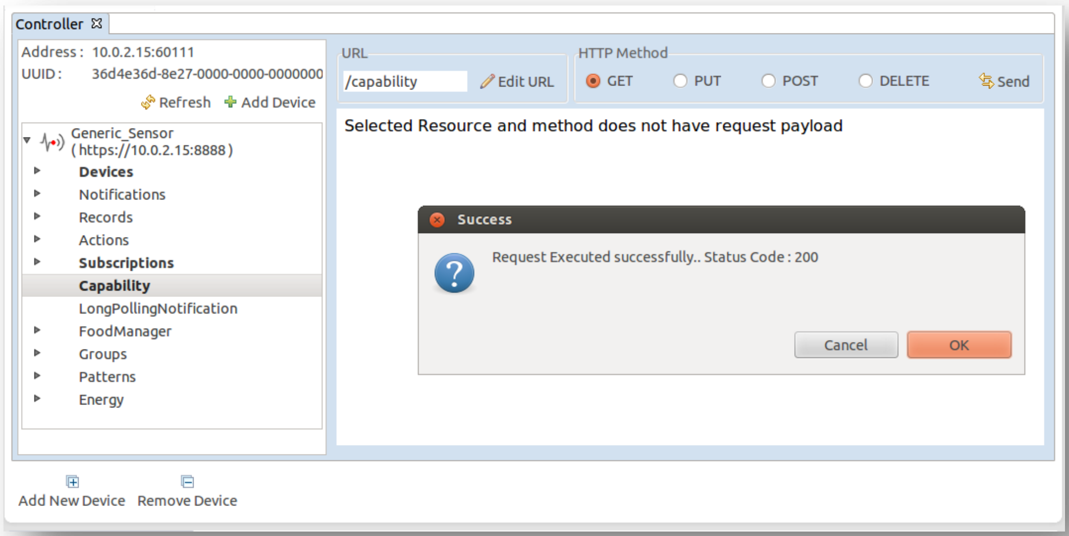
URL: /capability

HTTP Method: [✓ GET] PUT POST DELETE

Selected Resource and Method does not have request payload

NOTE: Please refer to section – 'Controller Test Tool' for sequence of steps to be followed for 'Controlling' resources (Request method as: PUT/POST/DELETE)

- Response from developed application (device) shall be 200, for example:



Address : 10.0.2.15:60111
UUID : 36d4e36d-8e27-0000-0000-0000000

Generic_Sensor
(https://10.0.2.15:8888)
Devices
Notifications
Records
Actions
Subscriptions
Capability
LongPollingNotification
FoodManager
Groups
Patterns
Energy

URL: /capability

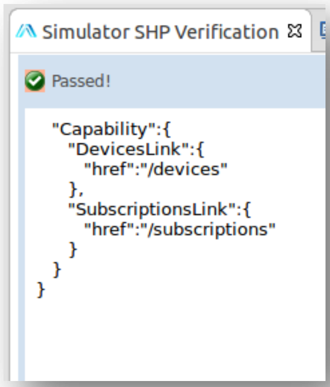
HTTP Method: [✓ GET] PUT POST DELETE

Selected Resource and method does not have request payload

Success

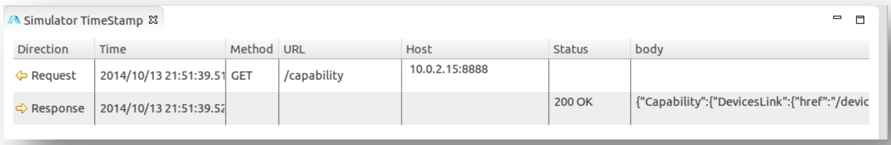
Request Executed successfully.. Status Code : 200

- Actual response (payload) for the request can be viewed on 'Simulator SHP Verification' panel and request and response details can also be viewed separately on 'Simulator TimeStamp' panel, for example:



Passed!

```
  "Capability":{
    "DevicesLink":{
      "href":"/devices"
    },
    "SubscriptionsLink":{
      "href":"/subscriptions"
    }
  }
}
```



Direction	Time	Method	URL	Host	Status	body
Request	2014/10/13 21:51:39.51	GET	/capability	10.0.2.15:8888		
Response	2014/10/13 21:51:39.52				200 OK	{"Capability":{"DevicesLink":{"href":"/devic

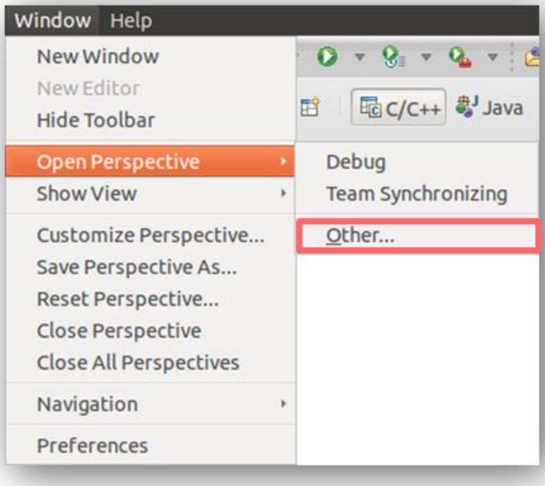
This finishes testing (discovery, monitor, and access) of sample '*Controlled*' application (Generic Sensor) using '*Controller*' simulators on SHP-SDK Simulator perspective.

3. Test Tool Usage

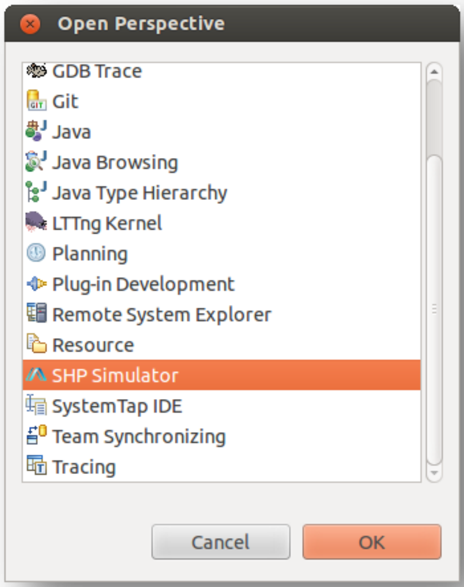
SHP Eclipse plug-in provides simulators which simulates the behavior of real SHP devices. Developer can use these simulators to test their application. Controller application developer can use Controlled Simulators for testing. Also Controlled application developers can use Controller Simulator for testing. Use "*SHP Simulator perspective*" to manage simulators.

3.1. Launching SHP Test Tool

- Select Window → Open Perspective → Other

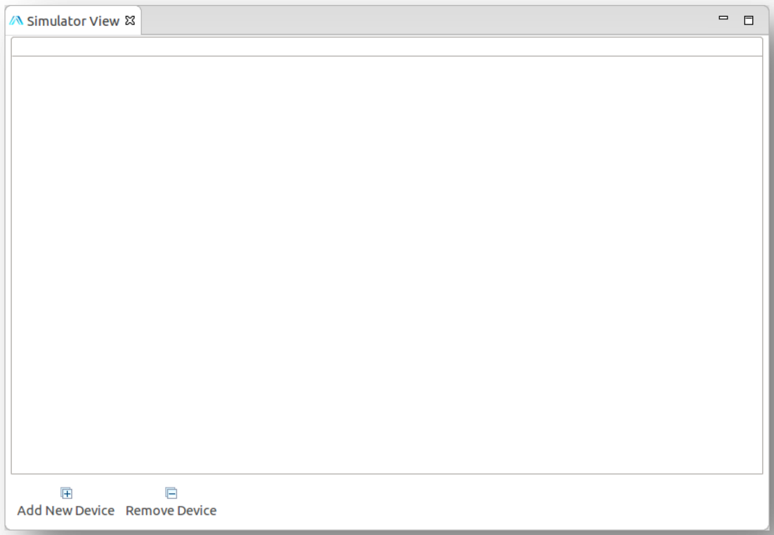


- Select "SHP Simulator"

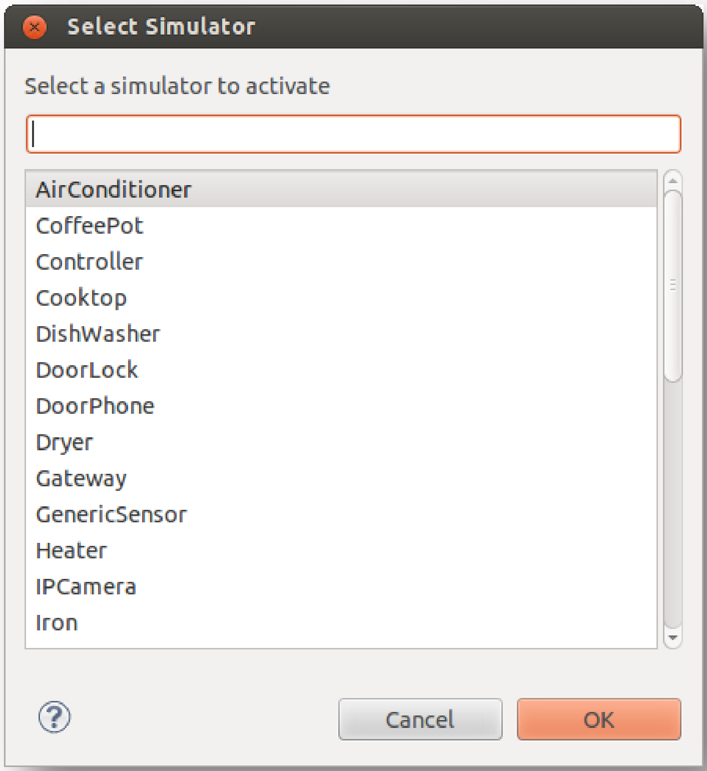


3.2. Controller Test Tool

- Click on ‘Add New Device’ button of ‘Simulator View’ to select and activate a SHP Simulator



- Select a Simulator to activate and press ‘OK’ button

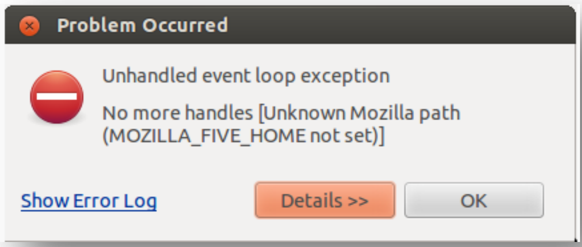

Select Simulator

Select a simulator to activate

AirConditioner
CoffeePot
Controller
Cooktop
DishWasher
DoorLock
DoorPhone
Dryer
Gateway
GenericSensor
Heater
IPCamera
Iron

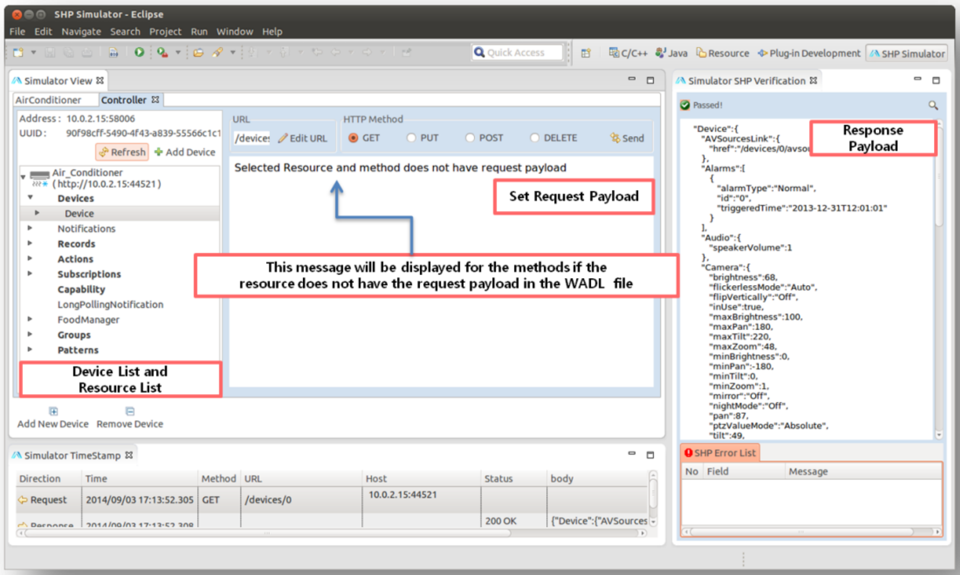
* When the Mozilla path error below occurs in Linux, install 'webkitgtk-1.0-0'.

: **sudo apt-get install libwebkitgtk-1.0-0**



```
Unhandled event loop exception

No more handles [Unknown Mozilla path
(MOZILLA_FIVE_HOME not set)]
```

- SHP Controller Simulator can detect SHP Controlled devices.
- Discovered devices will be shown in a tree view
- To Send a request,
    1. Select a device from device list
    2. Select a resource and method to execute
    3. Set Request Payload, if request needs if any
    4. Press "Send" button to send the request
    5. Response payload will be displayed.



Address : 10.0.2.15:58006
UUID : 90f98cff-5490-4f43-a839-55566c1c1

(http://10.0.2.15:44521)

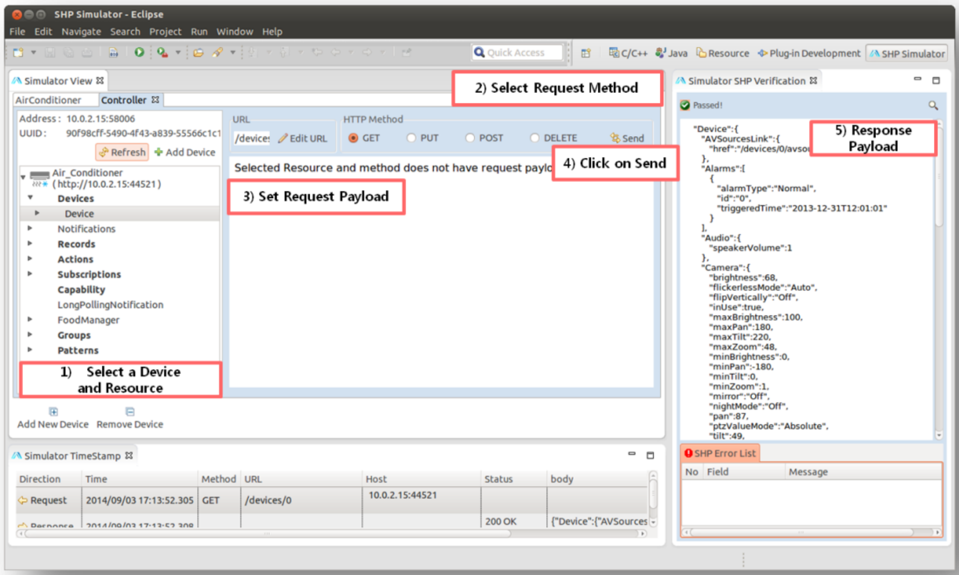
/devices

Selected Resource and method does not have request payload

```
"Device":{
  "AVSourcesLink":{
    "href":"/devices/0/avso
  },
  "Alarms":[
    {
      "alarmType":"Normal",
      "id":"0",
      "triggeredTime":"2013-12-31T12:01:01"
    }
  ],
  "Audio":{
    "speakerVolume":1
  },
  "Camera":{
    "brightness":68,
    "flickerlessMode":"Auto",
    "flipVertically":"Off",
    "inUse":true,
    "maxBrightness":100,
    "maxPan":180,
    "maxTilt":220,
    "maxZoom":48,
    "minBrightness":0,
    "minPan":-180,
    "minTilt":0,
    "minZoom":1,
    "mirror":"Off",
    "nightMode":"Off",
    "pan":87,
    "ptzValueMode":"Absolute",
    "tilt":49,
```

No	Field	Message

Direction	Time	Method	URL	Host	Status	body
Request	2014/09/03 17:13:52.305	GET	/devices/0	10.0.2.15:44521		
Response	2014/09/03 17:13:52.308				200 OK	{"Device":{"AVSources



Address : 10.0.2.15:58006
UUID : 90f98cff-5490-4f43-a839-55566c1c1

(http://10.0.2.15:44521)

Selected Resource and method does not have request paylo

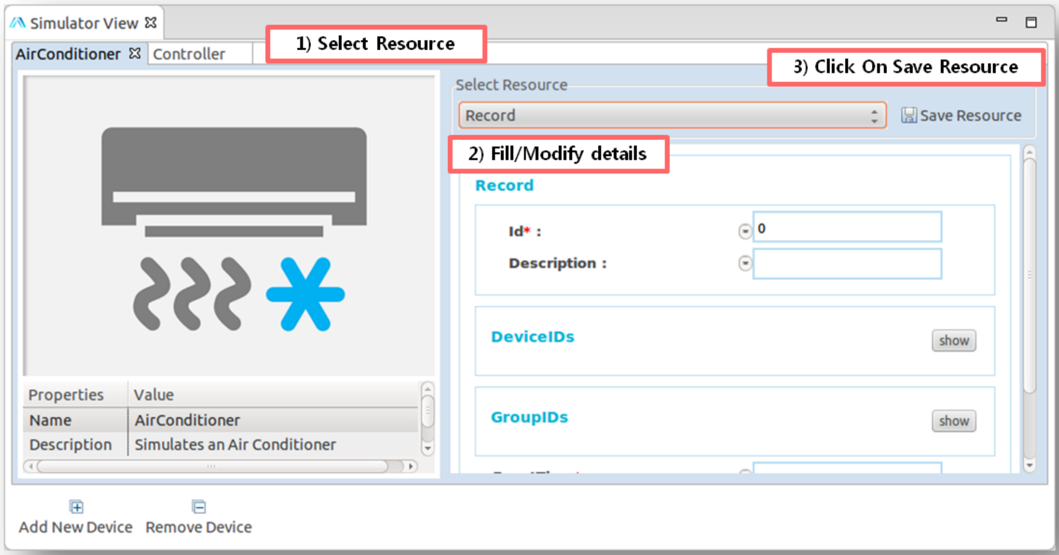
```
"Device":{
  "AVSourcesLink":{
    "href":"/devices/0/avso
  },
  "Alarms":[
    {
      "alarmType":"Normal",
      "id":"0",
      "triggeredTime":"2013-12-31T12:01:01"
    }
  ],
  "Audio":{
    "speakerVolume":1
  },
  "Camera":{
    "brightness":68,
    "flickerlessMode":"Auto",
    "flipVertically":"Off",
    "inUse":true,
    "maxBrightness":100,
    "maxPan":180,
    "maxTilt":220,
    "maxZoom":48,
    "minBrightness":0,
    "minPan":-180,
    "minTilt":0,
    "minZoom":1,
    "mirror":"Off",
    "nightMode":"Off",
    "pan":87,
    "ptzValueMode":"Absolute",
    "tilt":49,
```

Direction	Time	Method	URL	Host	Status	body
Request	2014/09/03 17:13:52.305	GET	/devices/0	10.0.2.15:44521		
Response	2014/09/03 17:13:52.308				200 OK	{"Device":{"AVSources

3.3. Controlled Test Tool

SHP Controlled Simulator simulates the behavior of a SHP controlled device. Controlled Simulator provides UI to configure current values of a resource. Simulator uses these values as response to requests received.

- To configure a resource
    1. Select a resource
    2. Fill the required details
    3. Click on Save to save details



Record

Id* : 0
Description :

Properties	Value
Name	AirConditioner
Description	Simulates an Air Conditioner

3.4. Easy Setup, registration, and Remote Access Test Tool

SHP Simulators (both Controlled and Controller) support simulation of complete Easy Setup routine along with Remote Access feature. Applications can make use of these simulators for verifying their Easy Setup or Remote Access specific functionality (either for Controller or Controlled).

Easy Setup process mainly comprises of three phases, **'Easy Setup'**, **'Registration'**, and '**SCS Login**'.

In **'Easy Setup'** phase,

- ❖ *Controller* (helper) device discovers *Controlled* device (device which needs external help for registration)
- ❖ *Controller* device provisions *Controlled* device by sharing details of 'Home AP'.
- ❖ And in return, *Controlled* device sends its *device* information to *Controller* for registration with Smart Home Server (SHS)

In **'Registration'** phase,

- ❖ *Controller* (helper) sends necessary credentials (for example, *Authorization Code* and *E-mail ID*) to *Controlled* device
- ❖ Then *Controller* performs *Controlled* device registration with Smart Home Server (SHS) by using the device details which are already retrieved in **'Easy Setup'** phase
- ❖ Upon successful registration, SHS server will return an ID (*peerID*) to the *Controller*.
- ❖ Then, *Controller* device sends all necessary details (*peerID*, *peerGroupID*, *countryCode* and *etc.*) required for *Controlled* device to perform login with Samsung Smart Connectivity Server (SCS)

In **'SCS Login'** phase,

- ❖ Using collected details in above two phases *Controlled* device attempts SCS Login and update the same to *Controller* device, this marks completion of Easy Setup process. *Controlled* device makes use of these details for further initialization as well

In '**Remote Access**',

- ❖ Upon successful SCS login, using '**Remote Access'** feature *Controller* device can manage (monitor/control) all registered *Controlled* devices through remote channel.

Note:

1. **Currently, Remote Access feature cannot be tested from the same instance of Eclipse; they are expected to launch Eclipse application for each simulated device, i.e., Controller on one Eclipse application and Controlled device on another instance of Eclipse application.**
2. **Before launch the SHP Simulator with Remote Access feature enabled, 2.4.8. Remote Access Configuration Files must be set appropriately in advance.**

Following are the sequence of steps involved in simulation of Easy Setup process, in which a *Controller* simulator (which has already been registered with the cloud) device helps in *Controlled* simulator device provisioning (Home AP details) and eventually empower *Controlled* devices to register themselves to the cloud.

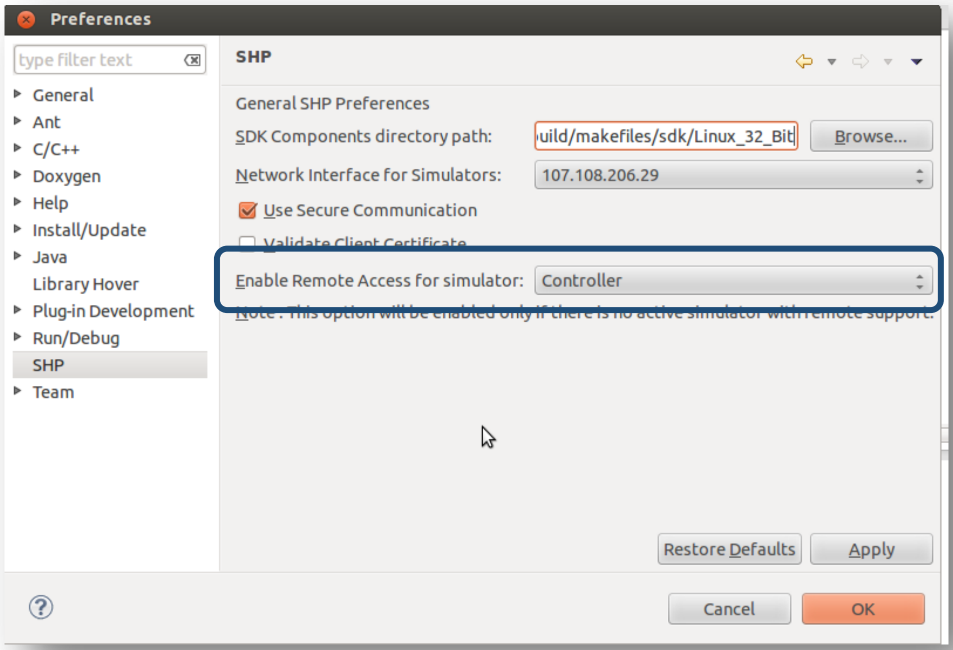
Note: Please refer to SHP-Architecture artifact for complete details on Easy Setup process

3.4.2. Step 1 – Enable Remote Access for *Controller* Simulator

As shown in the below figure, enable Remote Access for *Controller* simulator by selecting *Controller* for '**Enable Remote Access for simulator**' option on **Windows → Preferences** page.

Note 1: It is advised to rename current *Controller* eclipse configuration and save, so that the same configuration can be re-used for further launches. *If not, further launches of Controller Simulator do get different UUID*

Note 2: As mentioned in the below figure, one can enable Remote Access for *Controller* only if there is no active instance of *Controller* with Remote Access had already been enabled. At a time only one instance of device type simulator can have Remote Access being enabled.



SHP

General SHP Preferences

SDK Components directory path: uild/makefiles/sdk/Linux_32_Bit

Network Interface for Simulators: 107.108.206.29

[✓] Use Secure Communication

[ ] Validate Client Certificate

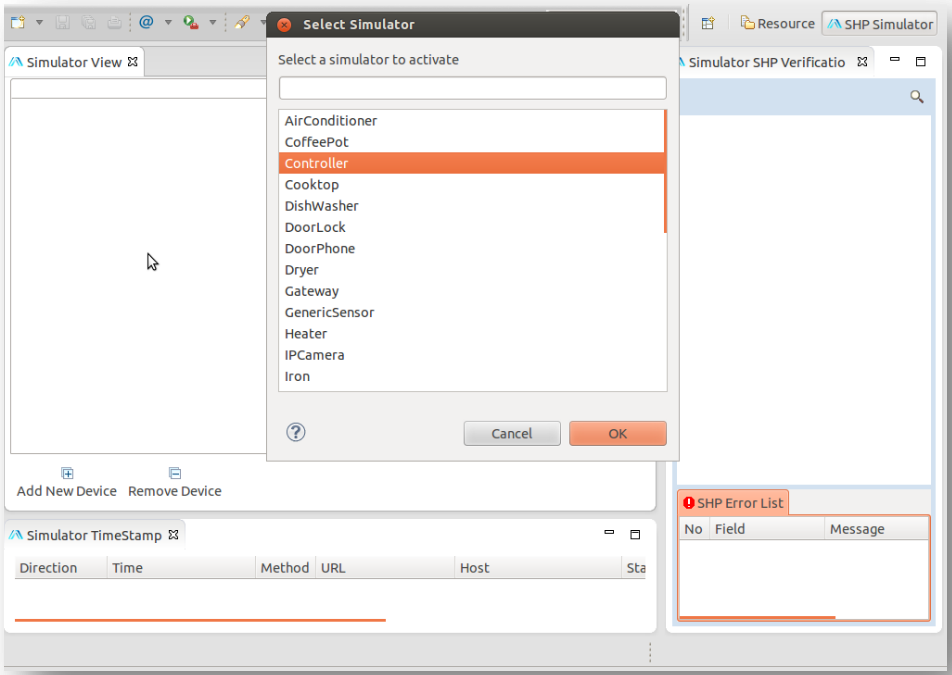
Enable Remote Access for simulator: Controller

Note : This option will be enabled only if there is no active simulator with remote support.

3.4.3. Step 2 – Launch/activate *Controller* Simulator

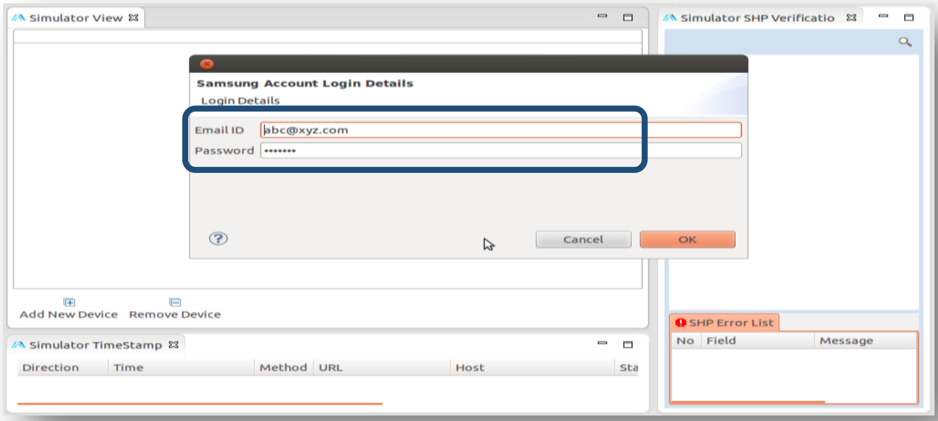
As shown in the below figure, launch/activate *Controller* simulator by clicking '*Add New Device*' button on '*Simulator View*' and by selecting *Controller* on '*Select Simulator*' window.

Note: Refer to Section 3.1 to launch '*SHP Simulator*'



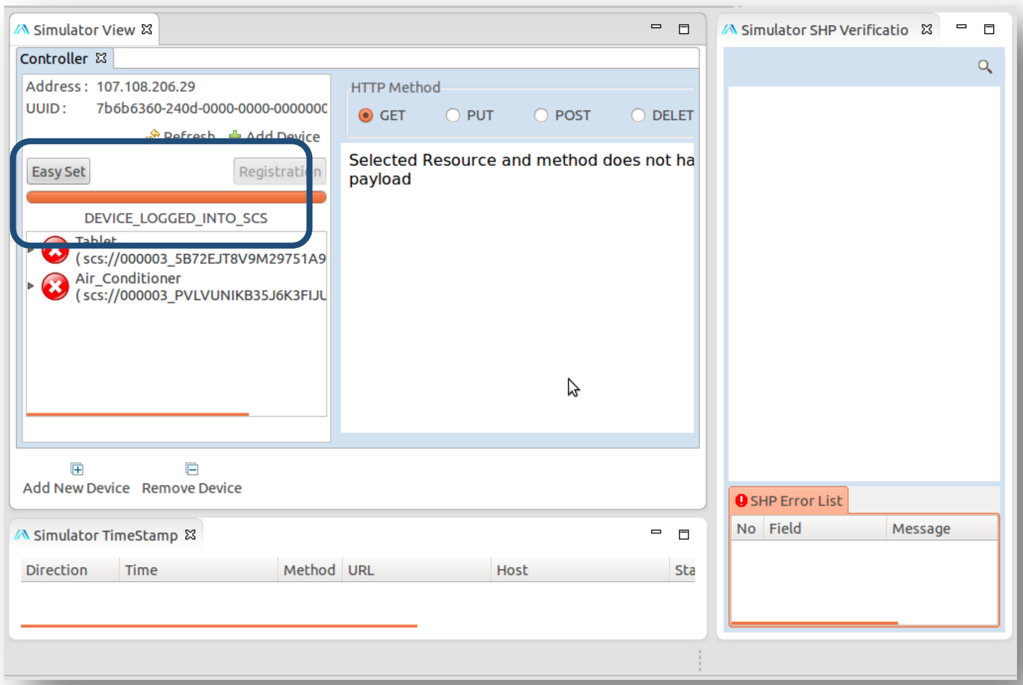
3.4.4. Step 3 – Provide User Credentials

As shown in the below figure, launch/activation of *Controller* simulator probes for user credentials (Samsung account), users are expected to key in *E-mail address* and *password* to be used for registering the device.



3.4.5. Step 4 – Ensure that *Controller* simulator is registered successfully

As shown in the below figure, ensure that *Controller* simulator got registered successfully, users shall verify that '**DEVICE_LOGGED_INTO_SCS**' has been shown on status bar. Also ensure that '*Easy Setup*' button has been enabled.



Address : 107.108.206.29
UUID : 7b6b6360-240d-0000-0000-00000

DEVICE_LOGGED_INTO_SCS

Tablet (scs://000003_5B72EJT8V9M29751A9
Air_Conditioner (scs://000003_PVLVUNIKB35J6K3FIJL

Selected Resource and method does not ha payload

Direction	Time	Method	URL	Host	Sta

No	Field	Message

3.4.6. Step 5 – Enable Remote Access for *Controlled* Simulator

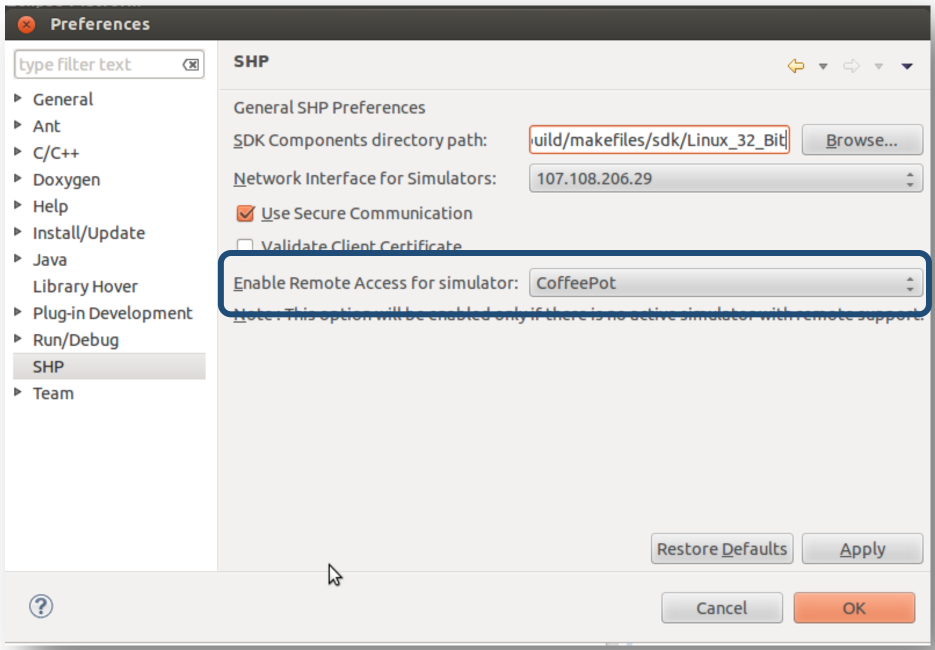
As shown in the below figure, now enable Remote Access for any *Controlled* simulator by selecting any one device type other than *Controller* for '**Enable Remote Access for simulator**' option on **Windows → Preferences** page.

Note 1: It is advised to rename current *Controlled* eclipse configuration and save, so that the same configuration can be re-used for further launches. *If not, further launches of Controlled Simulator do get different UUID*

Note 2: As mentioned in the below figure, one can enable Remote Access for *Controlled* (any device type) only if there is no active instance of same *device type simulator* with Remote Access had already been enabled. At a time only one instance of device type simulator can have Remote Access being enabled.

Note 3: For further explanation of Easy Setup, device type '*CoffeePot*' has been selected as *Controlled* Simulator.

This means using Easy Setup routine, *Controller* device simulator which had been successfully registered in Step 4 will help provisioning of *Controlled* Simulator ('*CoffeePot*').



Preferences

type filter text

- General
- Ant
- C/C++
- Doxygen
- Help
- Install/Update
- Java
- Library Hover
- Plug-in Development
- Run/Debug
- SHP
- Team

SHP

General SHP Preferences

SDK Components directory path: uild/makefiles/sdk/Linux_32_Bit [Browse...]

Network Interface for Simulators: 107.108.206.29

[✓] Use Secure Communication

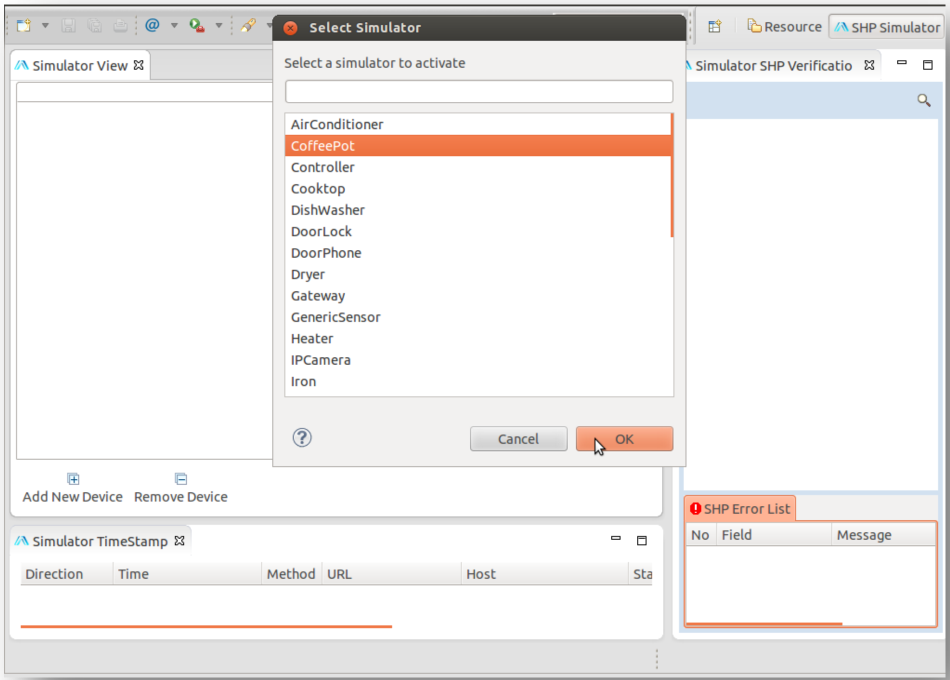
[ ] Validate Client Certificate

Enable Remote Access for simulator: CoffeePot

Note : This option will be enabled only if there is no active simulator with remote support.

3.4.7. Step 6 – Launch/activate *Controlled* (‘*CoffeePot*’) Simulator

As shown in the below figure, launch/activate *Controlled* simulator (‘*CoffeePot*’) by clicking ‘*Add New Device*’ button on ‘*Simulator View*’ and by selecting ‘*CoffeePot*’ on ‘*Select Simulator*’ window.



Select Simulator

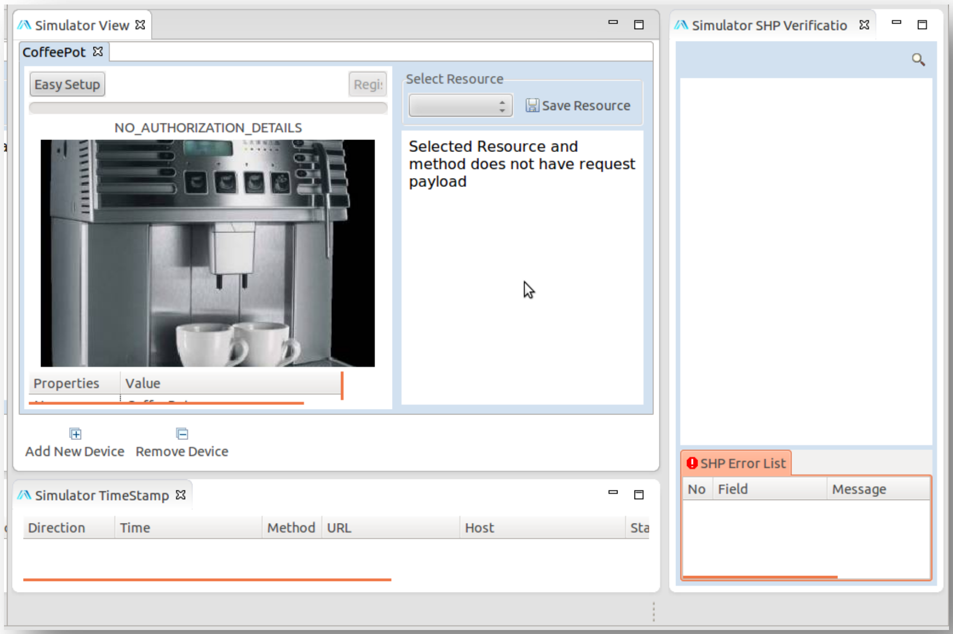
Select a simulator to activate

- AirConditioner
- CoffeePot
- Controller
- Cooktop
- DishWasher
- DoorLock
- DoorPhone
- Dryer
- Gateway
- GenericSensor
- Heater
- IPCamera
- Iron

3.4.8. Step 7 – Ensure that *Controlled* (‘*CoffeePot*’) simulator launched successfully but it is not registered

As shown in the below figure, ensure that *Controlled* (‘*CoffeePot*’) simulator has been launched successfully and it is not registered, users shall verify that ‘**NO_AUTHORIZATION_DETAILS**’ has been shown on status bar. *Controller* will help *Controlled* device getting necessary authorization details. And also ensure that ‘*Easy Setup*’ button has been enabled.

Note 1: If ‘**DEVICE_LOGGED_INTO_SCS**’ has been shown on status bar then users are expected repeat Steps 5 and 6 for any other device type.



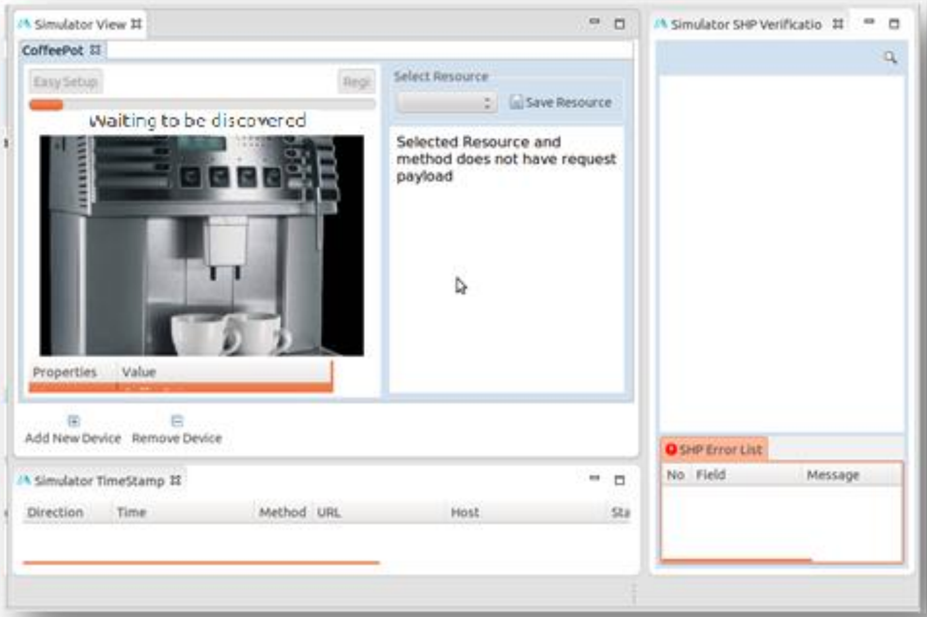
Selected Resource and method does not have request payload

3.4.9. Step 8 – Initiate Easy Setup process on *Controlled* (‘*CoffeePot*’) device simulator

For simulation of **‘Easy Setup’** process (for details refer to Section 3.4) all device simulators (*Controller* and *Controlled*) are provided with two buttons ‘*Easy Setup*’ and ‘*Registration*’. ‘*Easy Setup*’ button will initiate ‘**Easy Setup**’ phase and ‘*Registration*’ button will initiate ‘**Registration**’ phase of **‘Easy Setup’** process.

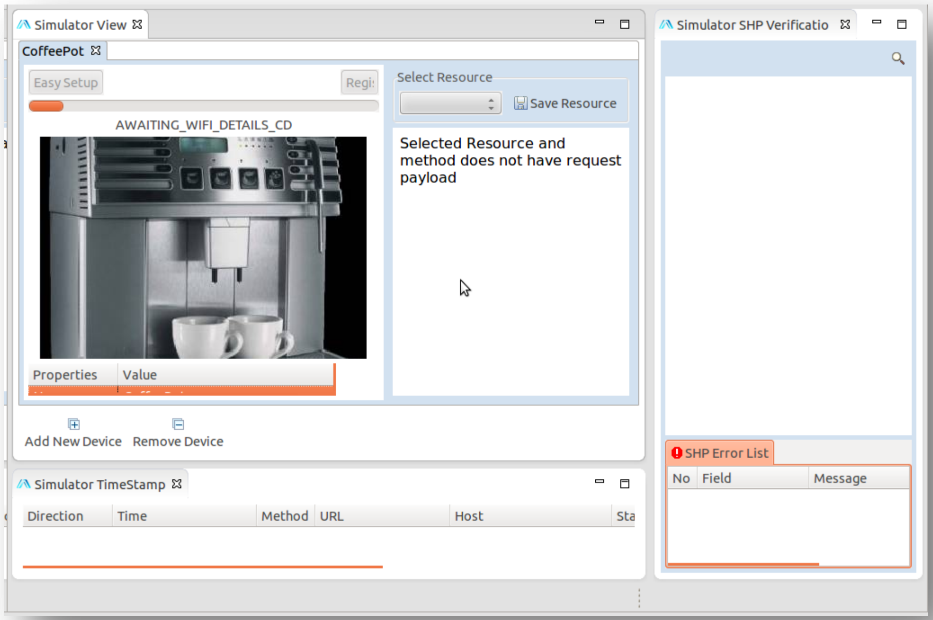
As shown in the below figure, first initiate ‘**Easy Setup**’ phase on *Controlled* (‘*CoffeePot*’) device simulator by clicking on ‘*Easy Setup*’ button present. Initiation of ‘**Easy Setup**’ phase can be ensured by checking status bar for:

If easy setup is not initiated at *Controller* side then look for ‘**Waiting to be discovered**’.



Selected Resource and method does not have request payload

If easy setup is initiated at *Controller* side (**Step 9**) after initiated at Controllable side, then look for '**AWAITING_WIFI_DETAILS_CD**'



Selected Resource and method does not have request payload

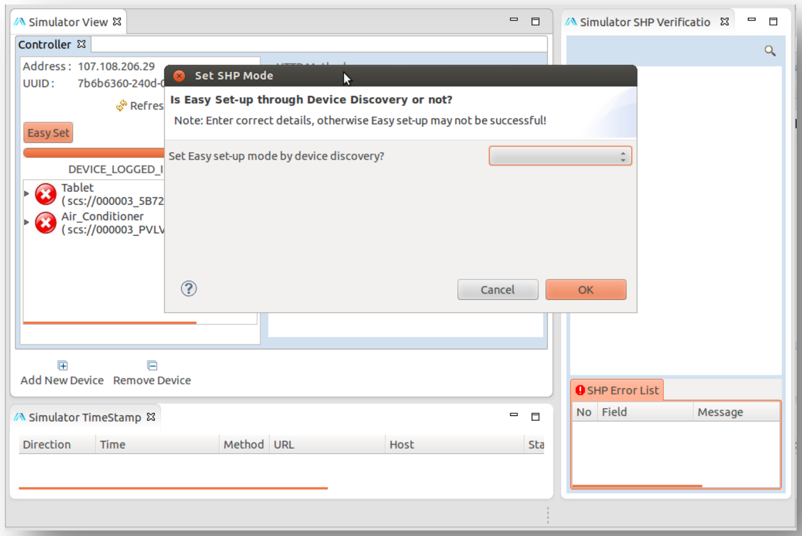
Note 1: Sequence/order of initiation is a must, initiation on *Controlled* device shall always be first and then only on *Controller*

Note 2: '**Easy Setup**' phase comprises of different stages (for details refer to Section 3.4), initially *Controlled* device does wait for Home AP (Wi-Fi) details from *Controller* device

3.4.10. Step 9 – Initiate Easy Setup process on *Controller* device simulator

‘**SHP Simulator**’ provides two ways of initiating ‘**Easy Setup**’ on *Controller* device. Users can either opt for Easy Setup through *Controlled* device discovery OR directly (without discovery).

After clicking ‘*Easy Setup*’ button on *Controlled* device, users are expected (**sequence/order is a must, initiation on *Controlled* device shall always be first and then only on *Controller***) to click same button on *Controller* (helper) device as well. As shown in below figure, users need to choose whether Easy Setup shall be initiated through *Controlled* device discovery OR directly (by selecting Yes – through discovery / No – for direct).



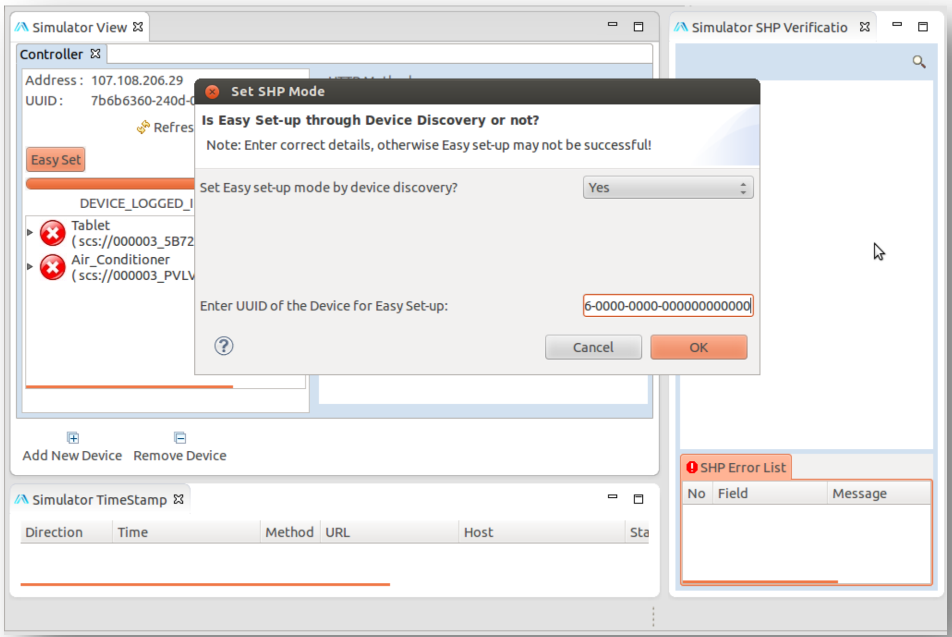
Set SHP Mode

Is Easy Set-up through Device Discovery or not?

Note: Enter correct details, otherwise Easy set-up may not be successful!

Set Easy set-up mode by device discovery?

As shown in below figure, selection of ‘**Yes**’ expects users to specify the UUID of non-registered *Controlled* device which needs to be discovered by the *Controller* device:


Set SHP Mode

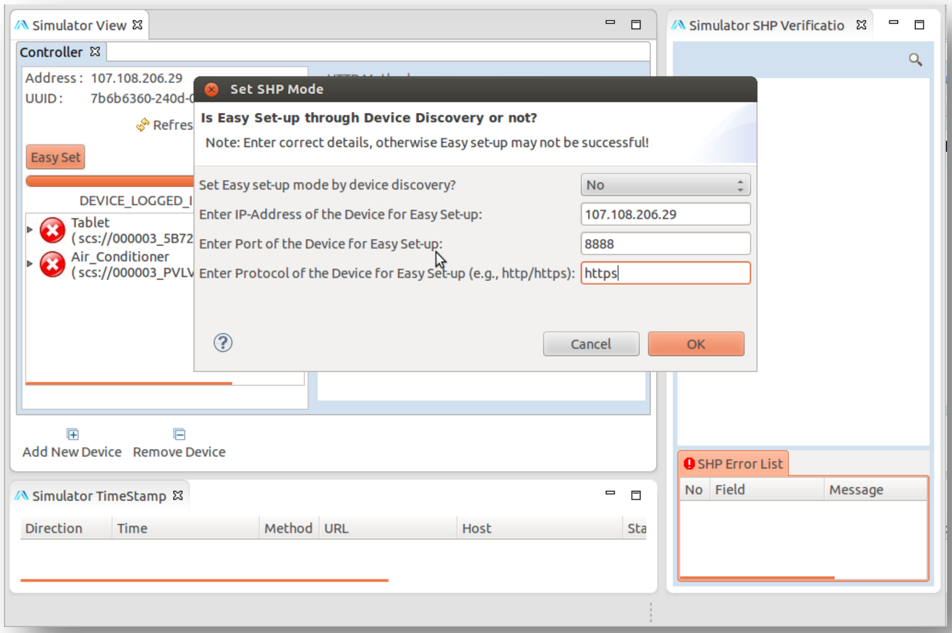
Is Easy Set-up through Device Discovery or not?

Note: Enter correct details, otherwise Easy set-up may not be successful!

Set Easy set-up mode by device discovery? Yes

Enter UUID of the Device for Easy Set-up: 6-0000-0000-000000000000

And as shown in below figure, selection of '**No**' expects users to specify details like IP-Address, Port, and Protocol (http/https) of the *Controlled* device which needs to be reached directly by *Controller* device:


Set SHP Mode

Is Easy Set-up through Device Discovery or not?

Note: Enter correct details, otherwise Easy set-up may not be successful!

Set Easy set-up mode by device discovery? No

Enter IP-Address of the Device for Easy Set-up: 107.108.206.29

Enter Port of the Device for Easy Set-up: 8888

Enter Protocol of the Device for Easy Set-up (e.g., http/https): https

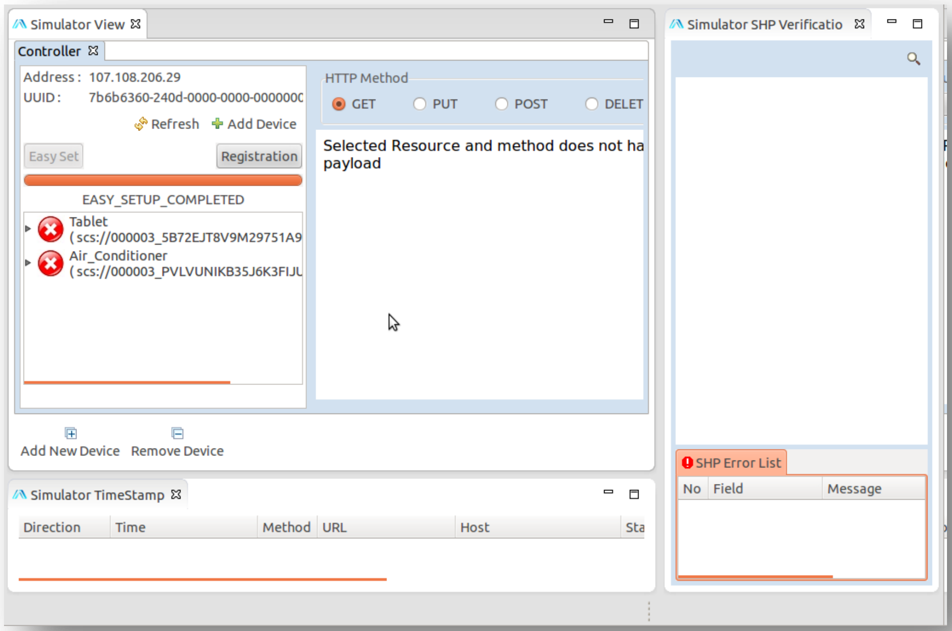
Based on above selection *Controller* device provisions *Controlled* device, if the selected option is through discovery then *Controller* device waits for the *Controlled* device to be discovered and then sends Wi-Fi details. Otherwise, it will directly sends without waiting for discovery.

Note for Application Development: *Devices which can support device discovery (SSDP) during Easy Setup phase can opt for 'Easy Setup through discovery'. Devices' which cannot support any sort device discovery needs to opt for 'Direct Easy Setup' option only.*

3.4.11. Step 10 – Ensure that Easy Setup phase is completed on *Controller* device Simulator

As shown in the below figure, ensure that '**Easy Setup**' phase has been completed successfully on *Controller* device simulator, users shall verify that '**EASY_SETUP_COMPLTED**' has been shown on status bar. And also ensure that '*Registration*' button has been enabled.

Note: '**EASY_SETUP_COMPLTED**' will not be displayed on status bar if what so ever reason the *Controlled* device is not discovered or not able to reached by *Controller* device. Refer to Section 3.4.19 for error details and respective display on status bar



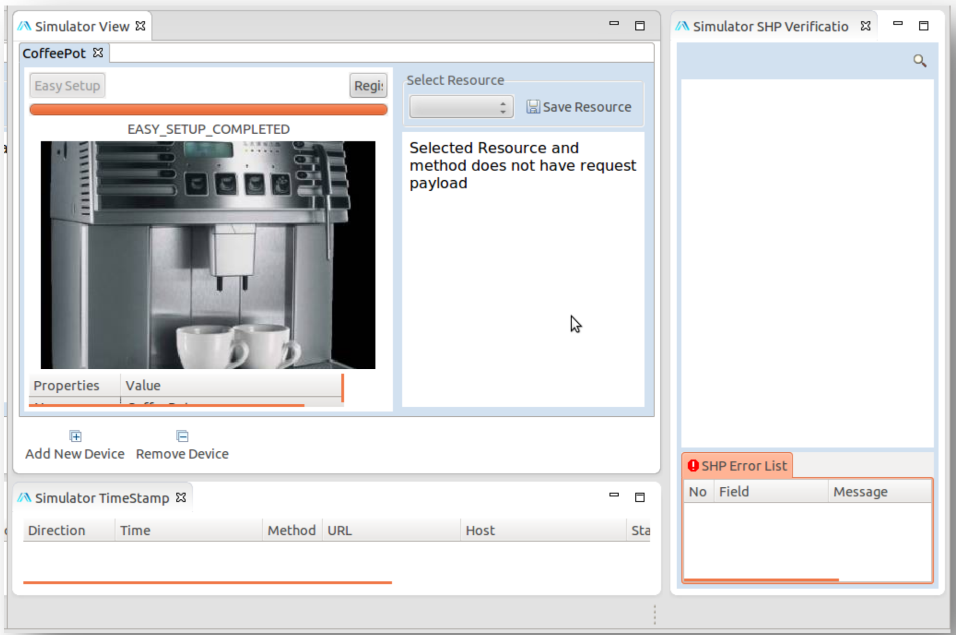
Selected Resource and method does not ha
payload

3.4.12. Step 11 – Ensure that Easy Setup phase is completed on *Controlled* ('*CoffeePot*') device Simulator

As shown in the below figure, ensure that '**Easy Setup**' phase has been completed successfully on *Controlled* ('*CoffeePot*') device simulator, users shall verify that '**EASY_SETUP_COMPLTED**' has been shown on status bar. And also ensure that '*Registration*' button has been enabled.

Note 1: '**EASY_SETUP_COMPLTED**' will not be displayed on status bar if what so ever reason the *Controlled* device is not discovered or not able to reached by *Controller* device. Refer to Section 3.4.20 for error details and respective display on status bar

Note 2: After completion of '**Easy Setup**' phase, 'Home AP – WiFi' details are available at *Controlled* device. Using these details *Controlled* device can connect to Home AP



Selected Resource and method does not have request payload

3.4.12.1. Steps for initiating and/or using 'Easy Setup' phase in SHP-Application Development:

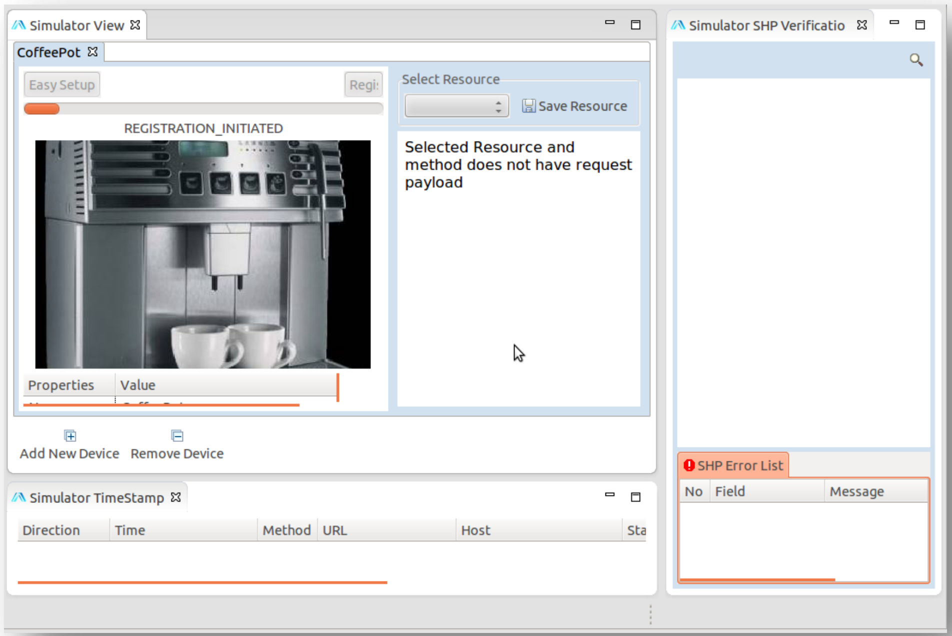
- Before setting SHP-Framework to 'Easy Setup' mode (*Sec::Shp::SHP::setSHPMode(EASY_SETUP_MODE)*), application has to ensure that Soft-AP (application needs to implement `SHPUtils::enableSoftAPMode()`) mode has been started on *Controlled* device and *Controller* should connect to the same Soft-AP
- In the Easy Setup mode, whenever *Controller* discovers *Controlled* device which is in Easy Setup mode through SSDP Device Discovery, it will send Home AP details by doing a PUT on ***/deivces/0/configuration/networks/0/wifi***
- Then, *Controlled* device sends its *device* information by doing POST on ***/devices/*** of *Controller* device
- Once both the devices exchange required information, then both will send ***onEasySetupModeCompleted()*** callback to the application, which will mark end of '**Easy Setup**' phase
- Once both *Controller* and *Controlled* device application receives ***onEasySetupModeCompleted()*** call back, both devices shall teardown their connection in Soft-AP mode (application needs to implement `SHPUtils::disableSoftAPMode()`) and connect back to the Home AP. Later, they are expected to initiate '**Registration**' phase by setting SHP-Framework mode to REGISTRATION_MODE

3.4.13. Step 12 – Initiate Registration phase on *Controlled* (‘*CoffeePot*’) device simulator

As shown in the figure below, upon successful ‘**Easy Setup**’ phase, initiate ‘**Registration**’ phase on *Controlled* (‘*CoffeePot*’) device simulator by clicking on ‘*Registration*’ button present. Initiation of ‘**Registration**’ phase can be ensured by checking status bar for ‘**REGISTRATION_INITIATED**’

Note 1: Sequence/order of initiation is a must, initiation on *Controlled* device shall always be first and then only on *Controller*

Note 2: ‘**Registration**’ phase comprises of different stages (for details refer to Section 3.4), initially *Controlled* device does wait for required credentials like *Authorization Code* and *E-mail ID* from *Controller* device



3.4.14. Step 13 – Initiate Registration phase on *Controller* device simulator

As shown in the below figure, upon successful ‘**Easy Setup**’ phase initiate ‘**Registration**’ phase on *Controller* device simulator (**sequence/order is a must, initiation on *Controlled* device shall always be first and then only on *Controller***) by clicking on ‘*Registration*’ button present. Initiation of ‘**Registration**’ phase can be ensured by checking status bar for ‘**REGISTRATION_INITIATED**’ and/or ‘**WAITING_FOR_HELPING_DEVICE_TO_BE_DISCOVERED_CR**’ and/or ‘**GETTING_AUTH_CODE_FROM_SERVER_CR**’.

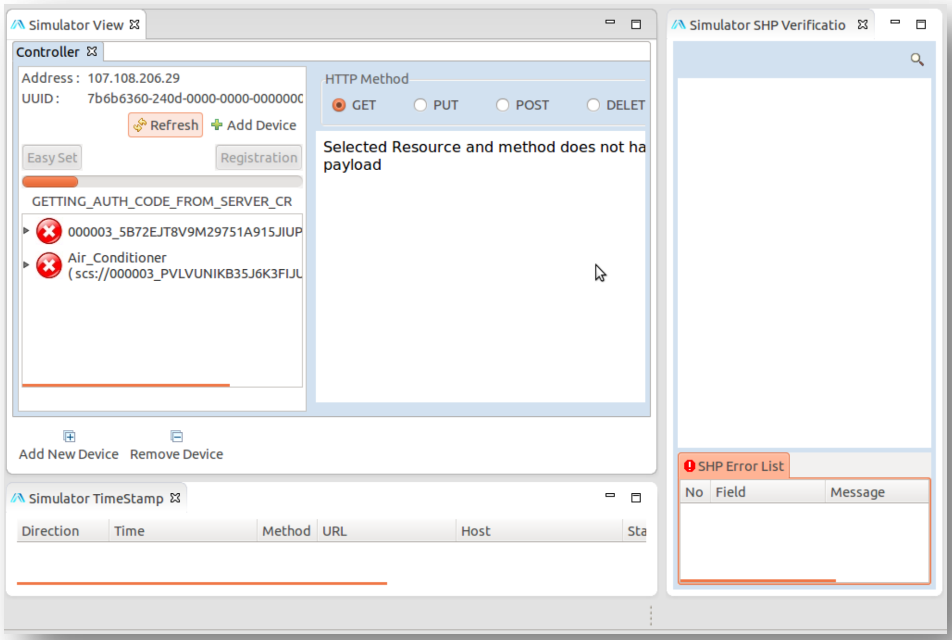
Note 1: ‘**Registration**’ phase comprises of different stages (for details refer to Section 3.4), initially *Controller* device wait for *Controlled* device to be discovered. And respective status on status bar would be ‘**WAITING_FOR_HELPING_DEVICE_TO_BE_DISCOVERED_CR**’,

Error Case 1: For what so ever reason, if *Controller* device fail to discover *Controlled* device then *Controller* device will continue to be in above state and eventually times out, refer to Section 3.4.20 for error details and respective display on status bar

Note 2: Once *Controlled* device is discovered, *Controller* device gets required credentials for *Controlled* device like *Authorization Code* and from Samsung Account Server using user credentials provided by user in Step and sends them to *Controlled* device

Error Case 2: For what so ever reason, if *Controller* device fail to get required credentials for *Controlled* device then *Controller* device will continue to be in one of the states (**DIFFERENT_COUNTRY_CODE, AUTH_CODE_EXPIRED,MISSING_MANDATORY_PARAMS,LOCAL_SERVER_ERROR, NO_AUTHORIZATION_DETAILS,CONNECTION_ERROR,REMOTE_SERVER_ERROR**) and eventually times out, refer to Section 3.4.20 for error details and respective display on status bar. Refer to API documentation for complete details on possible error cases during registration phase

Note 3: Please refer to **SHP-Architecture** artifact for completed details on '**Registration**' phase, and please refer to SHP-API Documentation for complete details on all possible Easy Setup notifications through '**Registration**' phase



Address : 107.108.206.29
UUID : 7b6b6360-240d-0000-0000-000000C

GETTING_AUTH_CODE_FROM_SERVER_CR

000003_5B72EJT8V9M29751A915JIUP

Air_Conditioner
(scs://000003_PVLVUNIKB35J6K3FIJL

Selected Resource and method does not ha
payload

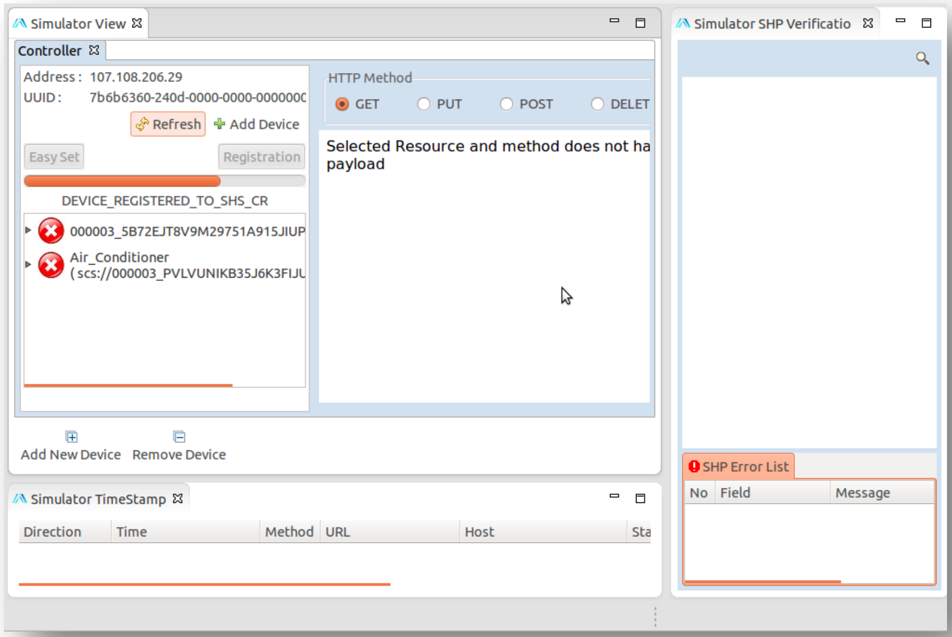
3.4.15. Step 14 – Ensure that Registration phase is completed on *Controller* device Simulator

As shown in the below figure, ensure that '**Registration**' phase has been completed successfully on *Controller* device simulator, users shall verify that '**DEVICE_REGISTERED_TO_SHS_CR**' has been shown on status bar.

Note 1: *Controller* device makes use of *device* details which are retrieved in '**Easy Setup**' phase and attempts registration with SHS Server

Note 2: Upon successful registration, SHS server will return an ID (*peerID*) to the *Controller* device, and then the *Controller* device sends all necessary details (*peerID*, *peerGroupID*, *countryCode* and *etc.*) required for *Controlled* device to perform login with SCS Server

Note 3: '**DEVICE_REGISTERED_TO_SHS_CR**' will not be displayed on status bar if what so ever reason the *Controller* device is not able to register *Controlled* device to SHS Server. Refer to Section 3.4.19 for error details and respective display on status bar



Selected Resource and method does not ha payload

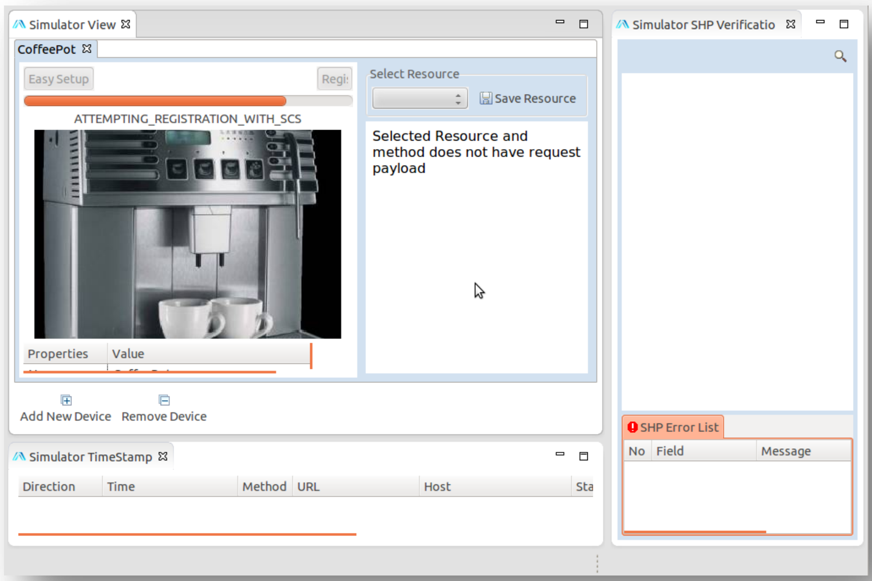
3.4.16. Step 15 – Ensure that Registration phase is completed on *Controlled* (‘*CoffeePot*’) device Simulator

As shown in the below figure, ensure that the ‘**Registration**’ phase has been completed successfully on *Controlled* (‘*CoffeePot*’) device simulator, users shall verify that ‘**ATTEMPTING_REGISTRATION_WITH_SCS**’ has been shown on status bar.

Note 1: *Controlled* device makes use of credentials like *Authorization Code* and *E-mail ID* from *Controller* device and attempts retrieving required access (*access token*) from Samsung Account Server

Note 2: Upon successful retrieval of access from Account Server, *Controlled* device to perform login with SCS Server using retrieved details (*peerID*, *peerGroupID*, *countryCode* and *etc.*) from *Controller* device

Note 3: ‘**DEVICE_REGISTERED_TO_SHS_CR**’ will not be displayed on status bar if what so ever reason the *Controller* device is not able to register *Controlled* device to SHS Server. Refer to Section 3.4.20 for error details and respective display on status bar

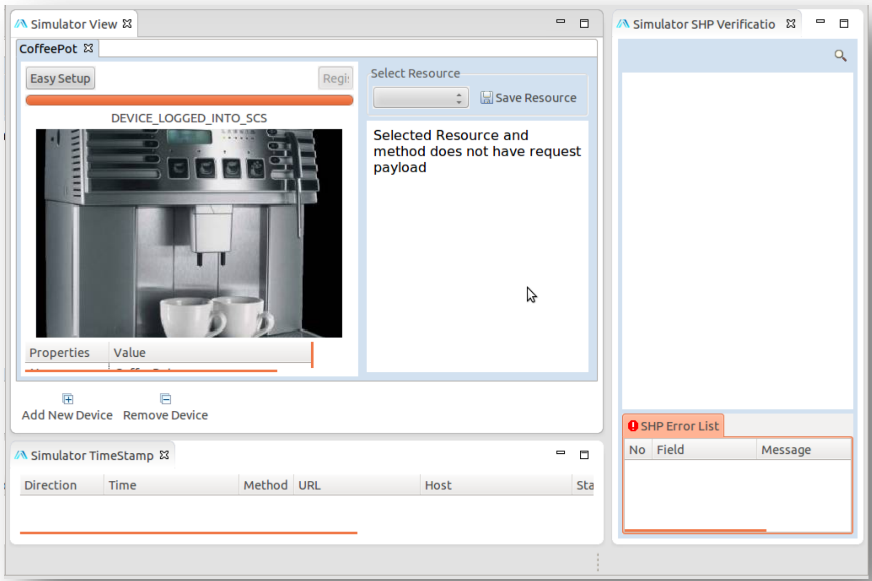


Selected Resource and method does not have request payload

3.4.17. Step 16 – Ensure that *Controlled* (‘*CoffeePot*’) device Simulator has been successfully logged into SCS Server

As shown in the below figure, ensure that *Controlled* (‘*CoffeePot*’) device has been successfully logged onto SCS Server, users shall verify that ‘**DEVICE_LOGGED_INTO_SCS**’ has been shown on status bar.

Note: ‘**DEVICE_LOGGED_INTO_SCS**’ will not be displayed on status bar if what so ever reason the *Controlled* device is not able to log onto SCS Server. Refer to Section 3.4.19 for error details and respective display on status bar



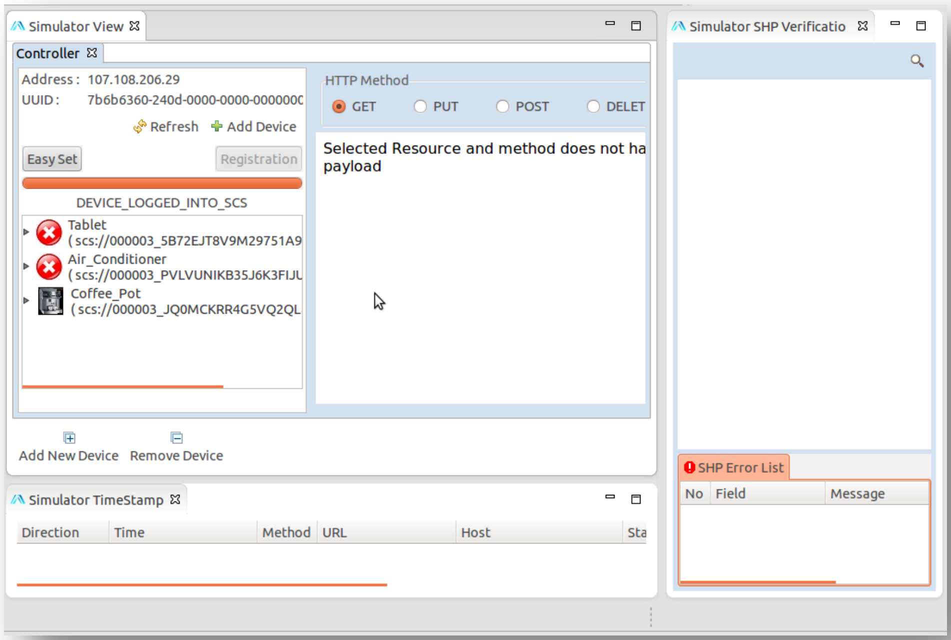
Selected Resource and method does not have request payload

3.4.18. Step 17 – Ensure that *Controller* device Simulator has been successfully logged into SCS Server after helping *Controlled* device for provisioning

As shown in the below figure, ensure that *Controller* device has been successfully logged onto SCS Server, users shall verify that '**DEVICE_LOGGED_INTO_SCS**' has been shown on status bar.

Note 1: '**DEVICE_LOGGED_INTO_SCS**' will not be displayed on status bar if what so ever reason the *Controller* device is not able to log onto SCS Server. Refer to Section 3.4.19 for error details and respective display on status bar

Note 2: Upon successful SCS login, *Controller* device shall see *Controlled* device amongst the list of the devices to be controlled



DEVICE_LOGGED_INTO_SCS

Tablet
(scs://000003_5B72EJT8V9M29751A9

Air_Conditioner
(scs://000003_PVLVUNIKB35J6K3FIJU

Coffee_Pot
(scs://000003_JQ0MCKRR4G5VQ2QL

Selected Resource and method does not ha
payload

3.4.18.1. Steps for initiating and/or using 'Registration' phase in SHP-Application Development:

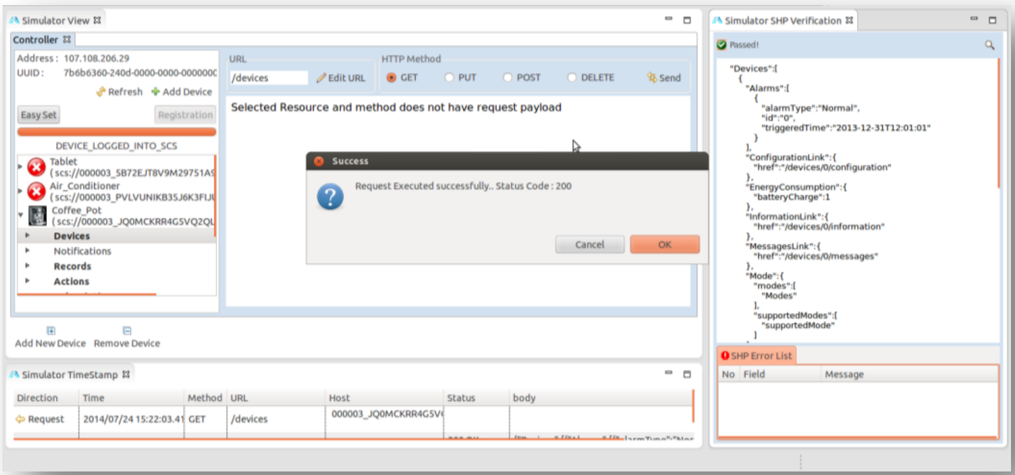
- Before setting SHP-Framework on both the devices (*Controlled* and *Controller*) to 'Registration' mode (*Sec::Shp::SHP::setSHPMode(REGISTRATION_MODE)*), application has to ensure that Soft-AP (application needs to implement `SHPUtils::disableSoftAPMode()`) mode has been tear down and both are connected to Home AP (application are expected to implement `SHPUtils::connectToHomeAccessPoint()`)
- In the Registration mode, *Controller* get *Authorization Code* (*authCode*) for *Controlled* device from Account Server
- Upon successful retrieval of *authCode*, it will send *authCode*, and *accountID* to *Controlled* device by doing a PUT on ***/deivces/0/configuration/remote***
- *Controlled* device attempts getting *Access Token* (*accessToken*) from Account Server using received *authCode*, and *accountID* from *Controller* device. *Controlled* device also informs status (success/failure) of the *accessToken* attempt as a response to PUT request on ***/deivces/0/configuration/remote***

- Upon successful PUT response, *Controller* device attempts registration on behalf of *Controlled* device with SHS Server by making use of *Controlled* device details which are retrieved during Easy Setup mode. For this, *Controller* device will do a POST on ***/shs/devices/*** to SHS Server
- Upon successful registration, SHS server will return an ID (*peerID*) to the *Controller* device, and then the *Controller* device sends all necessary details (*peerID*, *peerGroupID*, *countryCode* and *etc.*) required for *Controlled* device to perform login with SCS Server by doing a PUT on ***/deivces/0/configuration/remote***
- Upon successful reception of remote information, *Controlled* device (application is expected to make permanent store of this information) responds to the PUT request, this will mark completion of '**Registration**' phase
- Upon successful finish of '**Registration**' phase, both *Controller* and *Controlled* device automatically set themselves normal mode and attempt SCS Login / registration with all the required details
- Upon successful login to SCS Server will mark completion of **Easy Setup** process

3.4.19. Monitor / Control / Access Remotely – Remote Access Feature (sending a GET request through remote channel – SCS)

Remote Access feature of SHP allows *Controller* devices accessing / monitoring / control any *Controlled* device remotely through SCS channel.

Upon successful SCS Login, *Controller* device shall see *Controlled* device (registered) amongst the list of the devices to be controlled. And users shall be able to monitor / control / access these devices remotely by performing respective operation from *Controller* GUI. For example, select a *Controlled* device and initiate a GET request. Following figure displays the GET response from the *Controlled* ('*CoffePot*') device which is been registered above.



Address : 107.108.206.29
UUID : 7b6b6360-240d-0000-0000-000000C

DEVICE_LOGGED_INTO_SCS
Tablet (scs://000003_5B72EJT8V9M29751A
Air_Conditioner (scs://000003_PVLVUNIKB35J6K3FIJ
Coffee_Pot (scs://000003_JQ0MCKRR4G5VQ2QL
Devices
Notifications
Records
Actions

URL: /devices
HTTP Method: (● GET) PUT POST DELETE

Selected Resource and method does not have request payload

Success
Request Executed successfully.. Status Code : 200

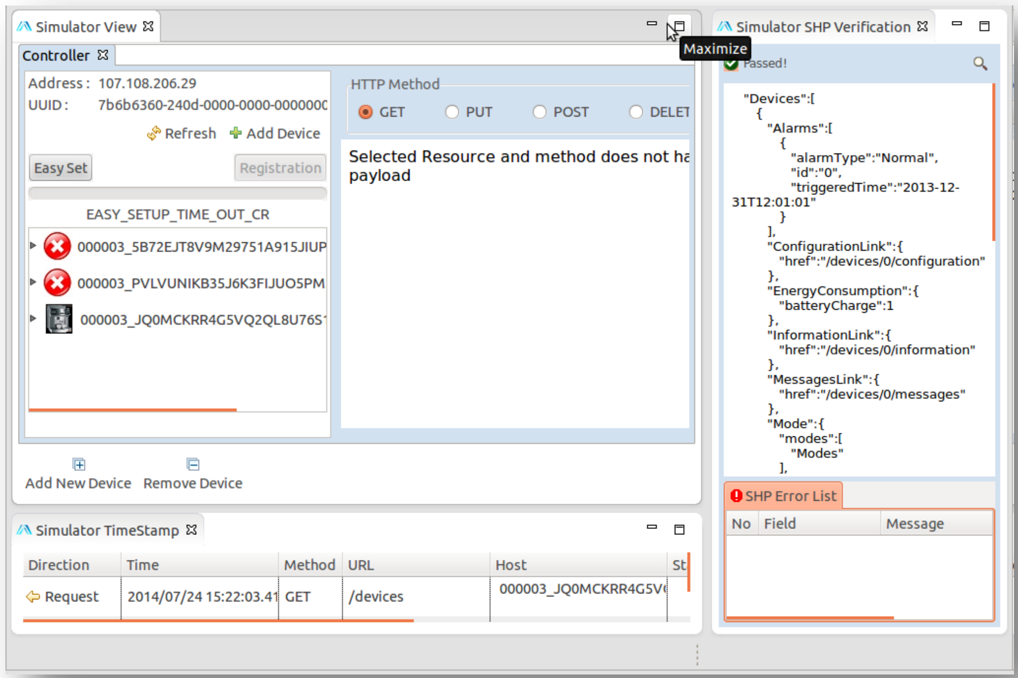
Direction	Time	Method	URL	Host	Status	body
Request	2014/07/24 15:22:03.41	GET	/devices	000003_JQ0MCKRR4G5V		

Passed!

```
"Devices":[
  {
    "Alarms":[
      {
        "alarmType":"Normal",
        "id":"0",
        "triggeredTime":"2013-12-31T12:01:01"
      }
    ],
    "ConfigurationLink":{
      "href":"/devices/0/configuration"
    },
    "EnergyConsumption":{
      "batteryCharge":1
    },
    "InformationLink":{
      "href":"/devices/0/information"
    },
    "MessagesLink":{
      "href":"/devices/0/messages"
    },
    "Mode":{
      "modes":[
        "Modes"
      ],
      "supportedModes":[
        "supportedMode"
      ]
```

3.4.20. Easy Setup Timer Timeout

SHP Framework initiates a timer on both the devices (*Controller* and *Controlled*) upon initiation of '**Easy Setup**' phase (Step 8 or Step 9). Default value of this timer is 300 seconds, and upon expiry of this timer SHP Framework will forcefully set the mode to NORMAL_MODE. Any error scenario during **Easy Setup** process (Easy Setup or Registration or SCL Login) will lead to expiry of the timer. And the same will be notified to the application as '**EASY_SETUP_TIME_OUT_CR**', same has been displayed in the following figure:



Address : 107.108.206.29
UUID : 7b6b6360-240d-0000-0000-000000C

EASY_SETUP_TIME_OUT_CR

- 000003_5B72EJT8V9M29751A915JIUP
- 000003_PVLVUNIKB35J6K3FIJUO5PM
- 000003_JQ0MCKRR4G5VQ2QL8U76S

HTTP Method: [✓ GET] PUT POST DELET

Selected Resource and method does not ha
payload

Direction	Time	Method	URL	Host	St
Request	2014/07/24 15:22:03.41	GET	/devices	000003_JQ0MCKRR4G5V	

```
"Devices":[
  {
    "Alarms":[
      {
        "alarmType":"Normal",
        "id":"0",
        "triggeredTime":"2013-12-
31T12:01:01"
      }
    ],
    "ConfigurationLink":{
      "href":"/devices/0/configuration"
    },
    "EnergyConsumption":{
      "batteryCharge":1
    },
    "InformationLink":{
      "href":"/devices/0/information"
    },
    "MessagesLink":{
      "href":"/devices/0/messages"
    },
    "Mode":{
      "modes":[
        "Modes"
      ],
```

No	Field	Message

3.5. Device Token

SHP mandates all *Controller* devices to be authorized by *Controlled* devices which it wants to access/monitor/control. SHP does D2D (Device to Device) authentication and authorization in different phases. For, SHP provides authorization by making use of the concept of issuing tokens to devices. All Smart Home *Controller* (Smart Phone/Tablet) devices are expected to get authorized by all Smart Home appliances (*Controlled* devices) by getting respective device tokens.

As per SHPs mandate all *Controlled* devices are expected to send error response (***WWW-Authenticate: Bearer error=”invalid_token”*** header**)** for any request from any *Controller* device which is not having proper authorization (correct device token). *Controller* device attempts getting authorization (device token) from each *Controlled* device right after finishing its D2D authentication.

SHP allows *Controller* devices to be authorized in two different ways:

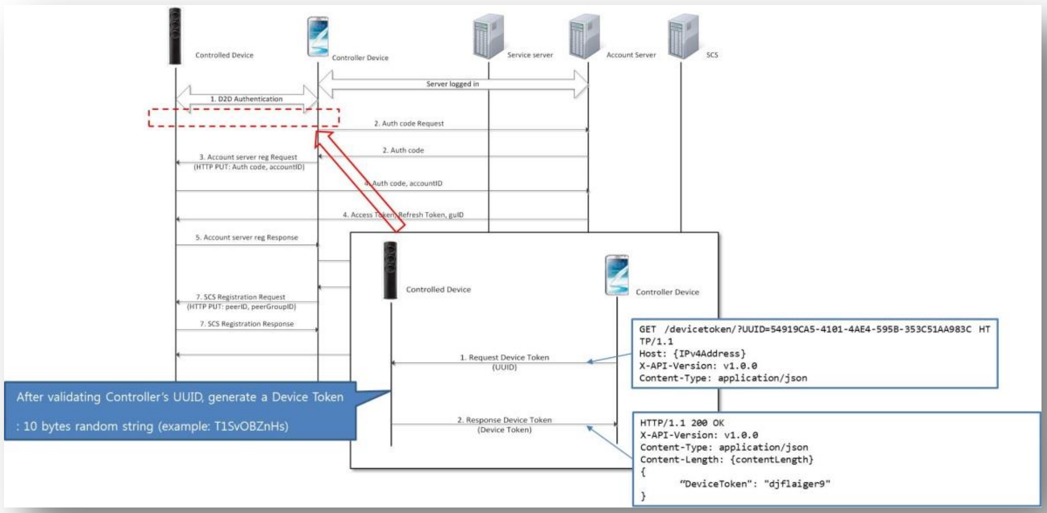
1. During **‘Registration’** phase of **Easy Setup** process, and another way
2. By sending device token request along with their UUIDs upon getting error response

3.5.1. Device Token issuance during ‘Registration’ phase of Easy Setup

Following are the steps involved in getting device token from *Controlled* device during **Easy Setup** process:

- ❖ Firstly, during **‘Easy Setup’** phase, after getting Home AP details from *Controller* device (*Step 6*) of Section 2.3.12.1 *Controlled* device requests device details (GET */devices*) of *Controller* device and stores UUID of the *Controller* device
- ❖ Then during **‘Registration’** phase, as a step next to D2D authentication, *Controller* device requests device token to the *Controlled* device (**GET /devicetoken/?UUID=<*UUID of controller*>**)

- Then the *Controlled* device validates the *Controller* device using UUID acquired during **'Easy Setup'** phase
- And upon successful of the UUID, *Controlled* device issues its device token as a response to the *Controller* device.


Controlled Device
Controller Device
Service server
Account Server
SCS
1. D2D Authentication
Server logged in
2. Auth code Request
2. Auth code
3. Account server reg Request
(HTTP PUT: Auth code, accountID)
4. Auth code, accountID
4. Access Token, Refresh Token, guID
5. Account server reg Response
7. SCS Registration Request
(HTTP PUT: peerID, peerGroupID)
7. SCS Registration Response
Controlled Device
Controller Device
1. Request Device Token
(UUID)
2. Response Device Token
(Device Token)
After validating Controller's UUID, generate a Device Token
: 10 bytes random string (example: T1SvOBZnHs)

```
GET  /devicetoken/?UUID=54919CA5-4101-4AE4-595B-353C51AA983C HT
TP/1.1
Host: {IPv4Address}
X-API-Version: v1.0.0
Content-Type: application/json
```

```
HTTP/1.1 200 OK
X-API-Version: v1.0.0
Content-Type: application/json
Content-Length: {contentLength}
{
      "DeviceToken": "djflaiger9"
}
```

Following is the excerpt from **Section 7.2** *of SHP-Architecture artifact which is the detailed procedure for* Device Token *issuance by* Controlled *device:*

Step 1: The *Controller* device requests device token to the *Controlled* device including the UUID of the *Controller* device.

Note: In order to issue the device token by the *Controlled* device, user consent (e.g., button pushing of the Controlled device or remote controller, etc.) is needed. If user consent is preceded in step 4 of Figure 4-4, additional user consent can be omitted within expiration timer after pushing the button. Expiration timer can be configured by the manufacturer of the *Controlled* device.

Step 2: The *Controlled* device compares the UUID from 'Device Token Request' message to the UUID acquired from the *Controller* device during Easy setup procedure specified in section 4.2.2. If both UUIDs are same, the *Controller* device is validated successfully

Step 3: If validation is done successfully, the *Controlled* device sends device token to the *Controller* device. Below is an example message of Local token requested response message

Note 1: During **'Registration'** phase, user consent is not required

Note 2: The Device token received from the *Controlled* device shall be included in all the request messages for validation by the *Controlled* device whether the Controller device is authorized or not. *Controlled* device responds to *Controller* device only if the device token in the request message is same with the device token which the *Controlled* device issued. Otherwise, *Controlled* device sends an error response – **"401 Unauthorized error"** with ***WWW-Authenticate: Bearer error="invalid_token"*** header

3.5.2. Reissuance of Device Token upon '401 Unauthorized error' – only for TV (only local)

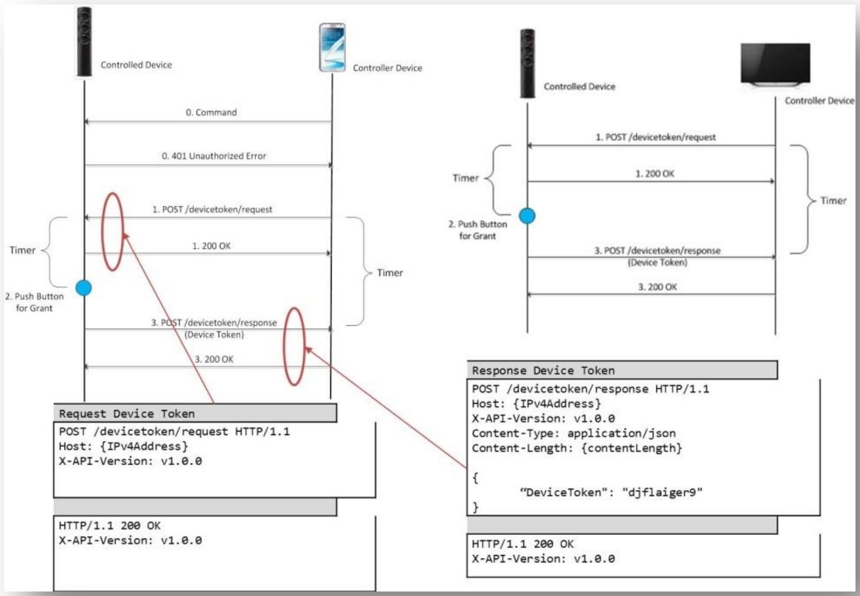
The Device token received from the *Controlled* device shall be included in all the request messages for validation by the *Controlled* device whether the Controller device is authorized or not. *Controlled* device responds to *Controller* device only if the device token in the request message is same with the device token which the *Controlled* device issued. Otherwise, *Controlled* device sends an error response: ***WWW-Authenticate: Bearer error="invalid_token"***

Upon reception of **"401 Unauthorized error"** from *Controlled* device, SHP allows another way of acquiring device token to *Controller* device from *Controlled* device, which is called getting authorized through reissuance of device token.

Note: However, as on today, this process is allowed only for TV (only local)

Reissuance of device token process is as follows:

- ❖ **Step 1:** Firstly, after getting error response, *Controller* device requests device token to the *Controlled* device (POST */devicetoken/request*),
  - ❖ **Note:** In this way of device token issuance, no UUID of the *Controller* device will be sent as part of request to the *Controlled* device and starts token wait timer
- ❖ **Step 2:** Upon reception of the device token request, *Controlled* device acknowledges it and straight away starts token wait timer and waits for user consent
  - ❖ **Note 1:** In order to issue the device token by the *Controlled* device, user consent (e.g., button pushing of the Controlled device or remote controller, etc.) is needed.
  - ❖ **Note 2:** Expiration timer can be configured by the manufacturer of the *Controlled* device
- ❖ **Step 3:** If user consent preceding timer expiry and is affirmative then the *Controlled* device sends device token to the *Controller* device
- ❖ **Step 4:** If user consent is not preceding timer expiration then the *Controlled* device sends error response to the *Controller* device



Request Device Token

```
POST /devicetoken/request HTTP/1.1
Host: {IPv4Address}
X-API-Version: v1.0.0
```

```
HTTP/1.1 200 OK
X-API-Version: v1.0.0
```

Response Device Token

```
POST /devicetoken/response HTTP/1.1
Host: {IPv4Address}
X-API-Version: v1.0.0
Content-Type: application/json
Content-Length: {contentLength}

{
        "DeviceToken": "djflaiger9"
}
```

```
HTTP/1.1 200 OK
X-API-Version: v1.0.0
```

3.5.3. Developer perspective of Device Token

Following is list of few of the functions which are involved in SHP-Application development related Device Token feature of SHP:

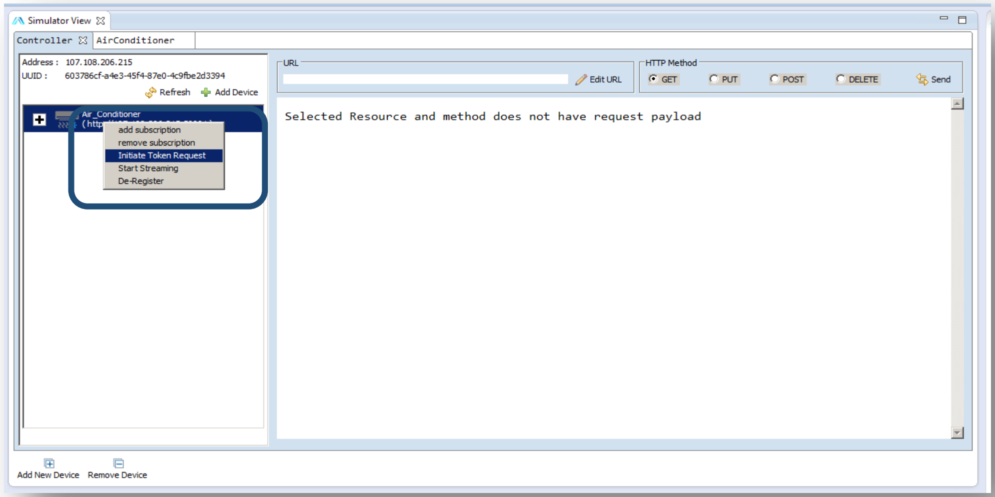
- `Sec::Shp::Device::initiateTokenRequestUsingUUID(const char *deviceUUID, bool fromSimulator)`, is the function which can be used by *Controller* application to initiate token request upon reception of error response from any device which it is interested in monitoring or controlling
  - **Developer Note:** Once Controller sends the device token request, then *Controller* application has to wait till either valid token received or request timed out.
    - Upon successful reception of device token, *Controller* application will get a callback (`SHPListener::updateUUIDAndTokenMap()`) from the framework
    - Upon failure in getting device token, *Controller* application will get "**DEVICE_TOKEN_REQUEST_TIMEOUT**" notification from the framework
- `SHPListener::tokenRequestNotification(std::string uuid)`, is the function which will be an application call back by SHP-Framework upon reception of device token request from *Controller* along with *Controllers* UUID. Application is expected either to Grant or to revoke permission for the request:
  - `::MySHPDevice::getInstance()->setTokenGrantStatus(bool true_false)` is the function which needs to called by the application based on users consent
    - `setTokenGrantStatus(true)` allows *Controlled* device to stop its token wait timer and send its device token to the caller as a response
    - `setTokenGrantStatus(false)` forces *Controlled* device to stop its token wait timer and send error response to the caller
- `SHPListener::getMyDeviceToken()`, is the method which will be an application call back by SHP-Framework to know and store *Controlled* devices' device token
  - **Note:** This function is applicable only for *Controlled* device application
- `std::string SHPListener::getUUIDAndTokenMap()`, is another method which will be an application call back by SHP-Framework, which will be called by *Controller* framework during its initialization.
  - Application developers are expected to implement this function to retrieve and return list of device (*Controlleds*) token and their mapped UUIDs as a string back to framework
- Similarly, `void SHPListener::updateUUIDAndTokenMap(uuid, deviceToken)`, is the call back from *Controllers* SHP-framework to the application upon reception of any *Controlled* device token
  - Application developers are expected to implement this function to store passed device token mapped to UUID to a permanent storage, which needs to be retrieved later
  - **Note:** Without this implementation, *Controller* application needs to go through complete process mentioned in Section 3.5.1 or 3.5.2 to get device token of *Controlled* device

Note: Please refer to SHP-API documentation for complete list of APIs related Device Token feature of SHP

3.5.4. Device Token Test Tool

SHP-Simulator supports simulation of both the device token issuance procedures mentioned in Section 3.5.1 and 3.5.2. Following is the screen shot which describes SHP-Simulation for procedure mentioned in Section 3.5.2.

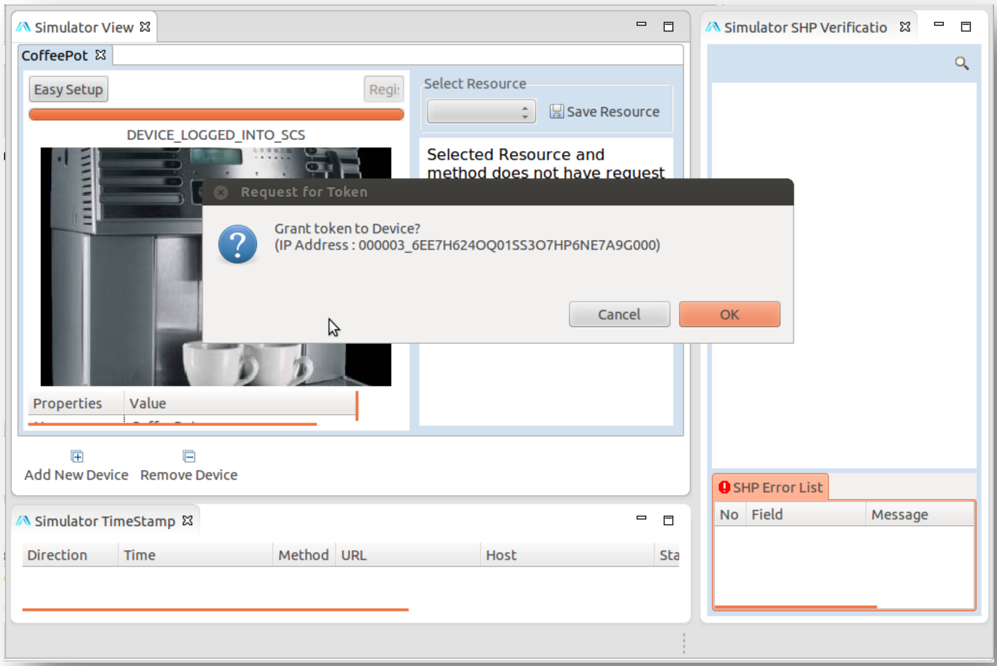
'Controller' simulation on SHP-SDK supports reissuance of Device Token upon '401 Unauthorized error', by providing an option to initiate token request. Users can initiate token request for a specific '*Controlled*' device is by right clicking on that device under discovered list and by selecting '**Initiate Token Request**' option.



Address : 107.108.206.215
UUID : 603786cf-a4e3-45f4-87e0-4c9fbe2d3394

```
Selected Resource and method does not have request payload
```

Note: In this mode, SHP-Simulator get users consent in form of a request popup, either to grant or revoke.



Selected Resource and method does not have request

Grant token to Device?
(IP Address : 000003_6EE7H624OQ01SS3O7HP6NE7A9G000)