Freedomotic Developer Manual

Release 5.6.0

Freedomotic Team

Apr 06, 2022

The Project

Everything you need to know about Freedomotic development.

Contents:

CHAPTER 1

What is Freedomotic?

Freedomotic is an open source, flexible and secure Internet of Things (IoT) development framework. It can be used to build and manage modern smart spaces. It is targeted at individuals (home automation) as well as businesses (smart retail environments, ambient aware marketing, monitoring and analytics, etc). Freedomotic can interact with well-known automation protocols as well as with "do it yourself" solutions. It treats the web, social networks and branded frontends as first class components of the system.

It allows you to build smart spaces. Freedomotic can manage many spaces, ranging from from small apartments to huge buildings, like museums, schools, corporate offices, malls and university campuses. For OEMs and software developers, Freedomotic is the solution to create building automation systems, smart retail environments, home automation managed services and innovative IoT ambient aware applications, drastically reducing development effort and time to market.

Freedomotic can be integrated with popular building automation technologies like BTicino OpenWebNet, Modbus RTU, Z-wave as well as custom automation projects using Arduino devices, do it yourself boards, third party graphical frontends, text to speech engines, motion detection using IP cameras stream, social networks, and much more... All this features can be delivered from a marketplace as downloadable plugins.

1.1 Vision

Bridging the gap between the physical and digital world; connecting people to things and value-added business services.

1.2 Mission

Developing an application framework, which reduces the effort and time to maket required to produce solutions based on the Internet of Things concept. This means we are making the environment aware of the people and the thinngs in it. Things can reach a new level of usefulness thanks to their new connected nature, allowing them to leverage the web and all of the information based services it provides.

1.3 Current development stage

The project is currently in an advanced beta stage. We are using the home automation segment to test and attract users but its range of application is much wider.

The final purpose of the project is to build a sort of **Content Management System (CMS)** for building automation. It will abstract and make easily available the common features required by building automation system in a way privates and companies can extend it to create custom context aware/environment aware services.

CHAPTER 2

Project History

The Department of Information Engineering and Computer Science (DISI) at the University of Trento had several research teams working on sensor networks for home monitoring and automation.

They needed a **framework** capable of easily integrating different projects developed in heterogeneous languages. This framework needed to make those projects work together, help simplify testing, and produce visual demos to show to research partners. Using a common framework, teams at DISI could focus on the core of their research instead of developing custom solutions for each individual project. The **Freedomotic** project was created to fulfill these needs.

The main goals and requirement of **Freedomotic** were: maintain a framework which was flexible and modular which could be easily integrated and adapted to different (and potentially unknown) needs, allowed for simple testing, and could produce visual demos.

These same goals and requirements continue to hold true today.

CHAPTER 3

Team

The Freedomotic team consists of a group of enthusiasts who work on the framework on a day to day basis.

Main contributors

Name	From	Contacts	Duties
Enrico Nicoletti	Italy	info@freedomotic.com	Project Founder, Project vision, Core Developer
Mauro Cicolella	Italy	mauro@freedomotic.com	Community Manager, Automation Protocols Integration, Marketing and end-users, Communication, Quality Assurance
Gabriel Pulido de Torres	Spain	gabriel@freedomotic.com	APIs Engineer, Mobile Platform Developer, Product Strategist, Technical Research, Development Workflow Manager
Alberto Mengoli	Italy	alberto@freedomotic.com	.Net Frontend Development, Testing, Usability feedback
Matteo Mazzoni	Italy	matteo@freedomotic.com	Core Developer, APIs Engineer, Plugins Developer, Technical Research

Table 3.1: Past members

Name	From	Duties
Niko Zarzani	Italy	UI Development
Roberto Socrates	Spain	Core Developer, Software architect

Many other developers have contributed to the project. Here the complete list.

CHAPTER 4

Features

Identity: Freedomotic allows each Thing to have a persistent unique identifier, which allows you to address it from all over the world, and it works no matter which automation protocol drives the Thing. You are safe from the protocols hell out there.

Services: Freedomotic is different, automations are not the end of the story, this framework is centred around the concept of Services for users, which may use automations to achive a goal. It is the Internet of Things at a new level. ThinG Wider!

Simulation: Freedomotic allows you to fully run it without any sensors or actuators connected. You can configure and test your automations before buying the hardware. This is great when planning a system with your customers, giving them a taste of the finished product.

Realtime Marketing: Freedomotic knows the environment topology (ie: rooms shapes and locations), the People and Things in it. This allows for users in the environment to be tracked and profiled, in addition to creating 1 to 1 realtime marketing campaigns. This feature is also great for disabled assistance and security focused systems.

Crosslanguage Rest API: Freedomotic allows you to control any aspect of the system with our JSON based REST APIs, from listing and controlling the Things in an environment, to retrieving, installing and managing plugins, all using familiar and developer-friendly technologies. The entire system is completely Events based. Components dialog together using text Events and Commands, so it's easy to integrate your ERP, CRM, or any legacy software you already run on your own premises. This is also great for building custom-branded frontends for web, mobile and desktop. In Freedomotic you can concurrently run as many frontends as you want, with the ability to have each one targeted to a specific audience.

Distributed: Freedomotic can run as a decentralized peer to peer network with no single point of failure, and can be deployed on a network of embedded systems like Raspberry Pi or on standard PCs and servers. For business this means you can have an instance running in the cloud connected to different satellites. You can manage the configuration and provide unique compute intensive features in the cloud (eg: face recognition) for a monthly fee.

Plugins: The system features are not hardcoded, and you can install new plugins at runtime. If you are interested in plugins development, take a look at our Developers Getting Started tutorial. Any developed plugin can be uploaded to an online marketplace (ours or your own) to allow 1-click installation.

Autodiscovery: Wouldn't it be great if you could turn on a light and have it automatically configured on the virtual environment map? Freedomotic can autodiscover the Things (eg: home automation devices) deployed in your real environment. No more diving in complex configuration files.

History Aware: Freedomotic can track any status changes in the environment and persist them in a database for analysis. For example, you can analyze consumption behaviors to implement energy-saving strategies, or you can learn more about how your customers interact with your shop by monitoring their visit patterns.

Secure, Multilanguage and Multiuser: Freedomotic is built ground up with multilanguage, multiuser and security features in mind. All these features come for free for each new plugin you develop, sparing a lot of time and effort. You can focus on your core business and let Freedomotic do the heavy lifting.

CHAPTER 5

Developers Quick Start

5.1 Requirements

- **Java JDK:** Version 8 OpenJDK/Oracle JDK (to install on Ubuntu: *sudo apt-get install openjdk-8-jdk*)
- **Maven:** Version 2 or 3 (to install on Ubuntu: *sudo apt-get install maven*)
- **Any OS** with java support (Linux, Windows, Mac, Solaris ...)

Development status: beta testing

Current released version: Freedomotic Commander 5.6 RC4 (released on 16 Aug 2017)

5.2 Set your local development environment

1. Fork Freedomotic on GitHub

- Create an account on https://github.com if you don't already have one
- Fork the Freedomotic repository by following this link: https://github.com/freedomotic/freedomotic/fork
- Create the local clone of your online fork with this command:

```
git clone https://github.com/YOUR-GITHUB-USERNAME/freedomotic.git
```

Now you are ready to work.

2. Enter the new local folder

```
cd freedomotic
```

3. Compile Freedomotic with maven

```
mvn clean install
```

4. **IMPORTANT!!!! THIS IS REQUIRED: copy the example-data folder into freedomotic-core/data.**

If you miss this step Freedomotic won't start

```
cp -r data-example/ framework/freedomotic-core/data
```

5. Run Freedomotic

```
java -jar framework/freedomotic-core/target/freedomotic-core/freedomotic.jar
```

5.3 Git repository is an SDK

The Git repository is a complete SDK with all you need to code and test your Freedomotic plugins. Once compiled for the first time, open the **freedomotic-core** project with your favourite IDE and start it to try Freedomotic.

To develop your own plugin, you can start from the **hello-world** example project included in *GIT_ROOT/plugins/devices/hello-world*.

Open it in your IDE, make some changes and compile. It will be automatically installed into the Freedomotic runtime (**freedomotic-core** project). Just start **freedomotic-core** to try your latest changes.

5.4 Create a build release

To create a new release package execute inside the ROOT folder

```
mvn clean install
```

The zip file containing the build release is located inside *GIT_ROOT/framework/freedomotic-core/target/release/*.

The release process is based on create-release.xml file.

5.5 Support

Please join our international or Italian community and share your experience.

CHAPTER 6

Maven quick reference sheet

Starting from Version 5.5, the Freedomotic build cycle is managed with Apache Maven. This quick reference explains how Maven phases are bound to specific tasks:

6.1 Priming build

First time compile, or to refresh the entire project and submodules

This will compile freedomotic-core and all basic plugins like base-objects, java-frontend, etc.

```
cd FREEDOMOTIC_ROOT
mvn clean install
```

6.2 How to start Freedomotic

You can do so from command line using

```
cd FREEDOMOTIC_ROOT
java -jar framework/freedomotic-core/target/freedomotic-core/freedomotic.jar
```

As an alternative, you can start **freedomotic-core** project from your favourite IDE.

6.3 Compile and test your own plugin

After doing changes to the plugin code...

This will compile your plugin and install it automatically into the Freedomotic runtime (**freedomotic-core**), ready to be started

```
cd FREEDOMOTIC_ROOT/plugins/devices/YOUR_PLUGIN_NAME
mvn clean install
```

6.4 Upload your own plugin on the marketplace

Share your own plugin in a convenient and easy to install way

This will compile your own plugin, deploy it to the online Maven repository and publish the new artifact on the related Freedomotic website marketplace page.

```
cd FREEDOMOTIC_ROOT/plugins/devices/YOUR_PLUGIN_NAME
mvn clean deploy -P market -D username="name" -D password="password"
```

More details on how to publish a plugin.

To know more about Maven phases, refer to the article "Maven: introduction to the lifecycle"

That's all! Open your favourite IDE and start the **freedomotic-core** project to run Freedomotic on your PC.

CHAPTER 7

Contributors workflow

A normal git development workflow should be used, with some considerations:

1. **Always** develop on a feature branch.
2. **Never** commit, rebase or merge changes into the master. Your forked **master** branch should just be a mirror of the official one and must be pulled frequently to be up to date with latest features and bugfixes.

If you don't follow the previous suggestions your pull request will probably be rejected.

As a recommendation, use meaningful branch names because they are going to be public.

7.1 How to contribute

Freedomotic follows the fork & pull process for collecting and quality-checking contributions from the development community.

This process works as follows:

1. You can start by forking our main git repository. This will create a Freedomotic fork named *YOUR-GITHUB-USERNAME/freedomotic.git*
2. Clone your fork locally by doing

```
git clone https://github.com/YOUR-GITHUB-USERNAME/freedomotic.git
```

3. On your local repository clone, create a branch with a meaningful name (e.g. new-feature-name). In case you are working to solve one of the known issues, please include the issue in the branch name (e.g. *fixing-Core-413*).

   Again, always develop on a branch.

4. When your proposed modifications are complete, you can generate a pull request, e.g. asking to merge *fixing-Core-413* into **freedomotic/master**. This can be done by publishing your new local branch online in your repository fork

```
git push origin BRANCHNAME
```

To generate a pull-request just click the **Create pull request** button on GitHub 6. Your pull request will be reviewed by the Freedomotic Development Team that will merge it into the main repository or ask for further revisions.

7.2 More Info

If you are clueless, the procedure described above is covered in full details with screenshots https://help.github.com/articles/using-pull-requests/#initiating-the-pull-request

CHAPTER 8

The 'Hello World' Plugin

Here you'll learn how to add features starting from the **hello-world** plugin. This plugin is made of some boilerplate Java code that you can use as base to create your own plugin.

Open the **freedomotic** maven project with your favourite IDE and compile it, if not already done. This will build a set of modules. Remember, the **freedomotic-core** module is the Freedomotic runtime, start it from your IDE to have it running.

Now open the **hello-world** module in your IDE and compile it. It will be automatically installed to the **freedomotic-core**.

8.1 Get familiar with Freedomotic

Here are some simple changes you can make with the plugin "hello-world".

8.1.1 Write a message to the GUI using a Freedomotic event

In the `onRun()` method of your plugin, write:

```
protected void onRun() {
  //get and format the current date and time
  DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
  Date date = new Date();
  //create a freedomotic message event
  MessageEvent message = new MessageEvent(null, "Hello world plugins says current␣
↪time is " + dateFormat.format(date));
  notifyEvent(message);
}
```

Then build your plugin, and start Freedomotic followed by the plugin. You will see a blue bubble with the current date and time at the upper left side of the environment map.

8.1.2 Make your plugin send emails

In the `onRun()` method of your plugin, write:

```
protected void onRun() {
    MessageEvent message # new MessageEvent(this, "The mail text");
    message.setType("mail"); //send this message as an email
    message.setTo("destination@gmail.com"); //the destination mail address
    notifyEvent(message);
}
```

Note: This requires the Mailer plugin to be installed and properly configured.

8.1.3 Print the list of things in the environment

In the `onRun()` method of your plugin, write:

```
protected void onRun() {
        StringBuilder buffer = new StringBuilder();
        for (EnvObjectLogic object : EnvObjectPersistence.getObjectList()) {
            buffer.append(object.getPojo().getName() + "\n");
            for (BehaviorLogic behavior : object.getBehaviors()) {
                buffer.append("  " + behavior.getName() + ": " + behavior.
↪getValueAsString()  + "\n");
            }
        }

        //print the string in the Freedomotic log using INFO level
        LOG.info(buffer.toString());
}
```

Then build your plugin, and start Freedomotic followed by the plugin. Right click on the **Log Viewer** plugin to see the list of things.

Note: Change the logging level to **INFO**, using the combobox of log viewer plugin, to filter out the less important log records.

8.1.4 Change a thing location on the map

This piece of code iterates over all loaded things and moves objects of type Person to a random location. The `randomLocation()` function should be implemented and must return a `com.freedomotic.model.geometry.FreedomPoint` type (remember to add freedomotic-model.jar to your classpath)

```
protected void onRun() {
    for (EnvObjectLogic anObject : EnvObjectPersistence.getObjectList()) {
        if (anObject instanceof it.freedomotic.objects.impl.Person){
            Person person = (Person)anObject;
            FreedomPoint location = randomLocation();
            person.getPojo().getCurrentRepresentation().setOffset(
                    (int)location.getX(),
                    (int)location.getY()
```

```
                    );
            person.setChanged(true);
        }
    }
}
```

8.1.5 Change things state programmatically

If you want to change the object state according to a value read from a web service, such as a weather forecast: https://github.com/freedomotic/freedomotic/wiki/Bound-objects-state-to-web-services-data

If you want to change the object state according to a value read from a hardware device, like an Arduino relay board: https://github.com/freedomotic/freedomotic/wiki/Bound-objects-state-to-hardware-data

8.1.6 Interact with users using a dialog box with multiple answers

You can take full advantage of other installed modules from your plugin. For example you can use a third party text to speech plugin to make it say something programmatically from your plugin.

You don't need to worry about how the external plugins works, you simply send to it a generic command. The example below uses the **Jfrontend** plugin to prompt a dialog with three choices.

```
public void askSomething() {
 final Command c = new Command();
 c.setName("Ask something silly to user");
 c.setReceiver("app.actuators.frontend.javadesktop.in");
 c.setProperty("question", "<html><h1>Do you like Freedomotic?</h1></html>");
 c.setProperty("options", "Yes, it's good; No, it sucks; I don't know");
 c.setReplyTimeout(10000); //10 seconds

 new Thread(new Runnable() {
   public void run() {
     Command reply = Freedomotic.sendCommand(c);
       if (reply != null) {
            String userInput = reply.getProperty("result");
              if (userInput != null) {
                System.out.println("The reply to the test question is " + userInput);
                 } else {
                     System.out.println("The user has not responded to the question␣
↪within the given time");
                    }
                 } else {
                    System.out.println("Unreceived reply within given time (10 seconds)
↪");
               }
            }
          }).start();
   }
```

8.1.7 Add a GUI to the plugin

To add a graphical interface, you must create a Jframe and link it to the plugin in `onStart()` with the following code:

```
gui = new PluginJFrame();
```

To open the GUI, right-click on the plugin icon.

CHAPTER 9

Architecture components

Freedomotic is composed by a core framework and plugins .

9.1 Framework

The core part is a **framework** that:

1. **implements** a language independent messaging system based on **Enterprise Integration Pattern**. So you can develop in your favorite language and just exchange messages with the other software components. The aim of the messaging system is to link all software modules together in a flexible and abstract way, relating them using the concept of channels (publish-subscribe to different levels of a topics hierarchy)

2. **maintains** an internal data structure representing the environment (topology, rooms connections as a graph, ...), the things in zones and their state (on, off, open, closed, 50% dimmed, ...)

3. **creates** an abstraction layer so users and external software modules can use a high level logic like `turn on kitchen light` instead of `send to COM1 port the string #*A01AON##`. This way a developer can leverage other plugins features at a high logical level because the modules can see the same environment map as the user. All data components (environment, objects, triggers, commands) can be defined in XML and easily exchanged on the network between different nodes of the P2P Freedomotic network

4. **provides** a rules engine coupled with a natural language processing system to let the user write automations in plain English like `if outside is dark turn on living-room light`. You can add, update and delete this automations at runtime using any human computer interface like GUIs, or even speak them.

9.2 Plugins

Freedomotic plugins create additional features to the core framework and can be developed and distributed as completely independent packages on the Freedomotic marketplace.

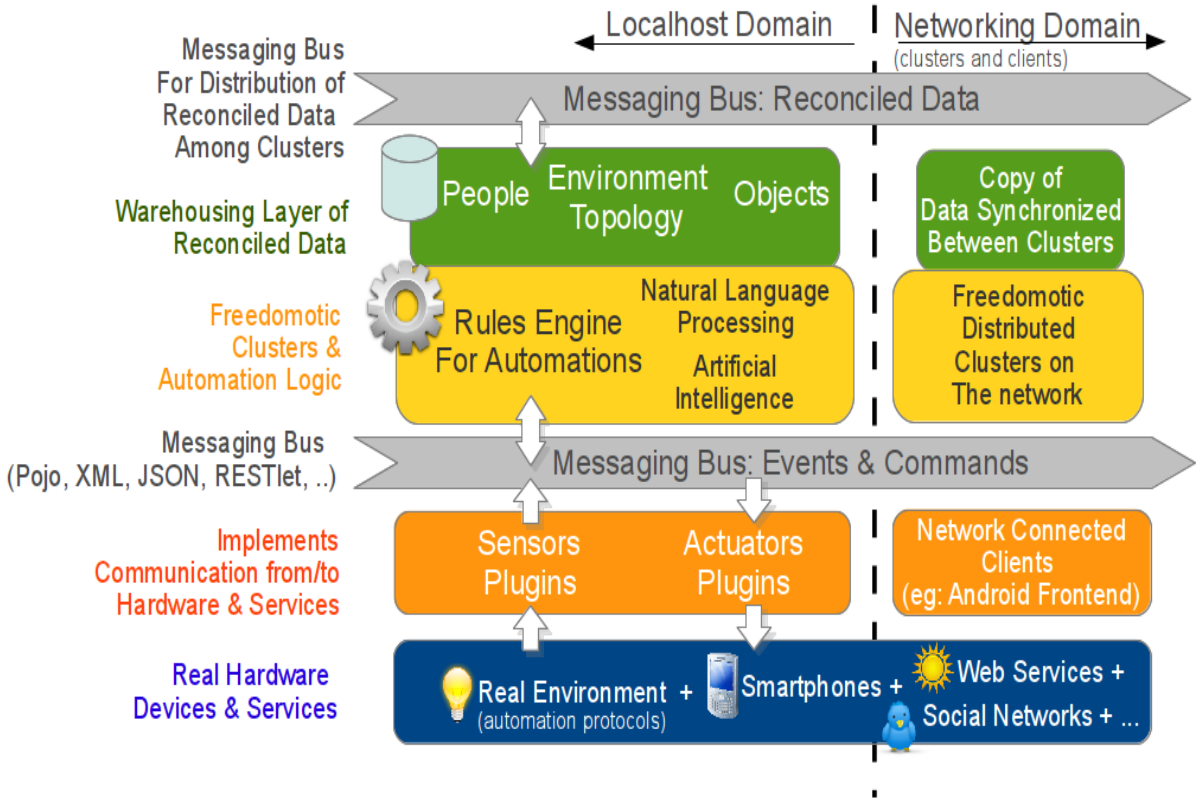


Fig. 9.1: Freedomotic architecture

9.2.1 Device plugins

Device plugins are generally developed to communicate with automation hardware like X10, KNX and so on, but also can provide graphical frontends and "web service readers" as Freedomotic plugins just as any other source of info, like webcams, text to speech engines and SMS senders can also be used.

9.2.2 Object plugins

Developers can also create object plugins which are pieces of software that models the behavior of objects like lamps, doors, etc in order to instruct the framework on how these objects should behave.

For example, a lamp object plugin tells the framework that a lamp has a boolean behavior called **powered** and a **dimmer** behavior which is represented by an integer from 0 to 100. A lamp can **turn on**, **turn off** and **dimm**. If **dimmer** = 0% the lamp **powered** behavior is set to false and if **dimmer** > 0% **powered** becomes true.

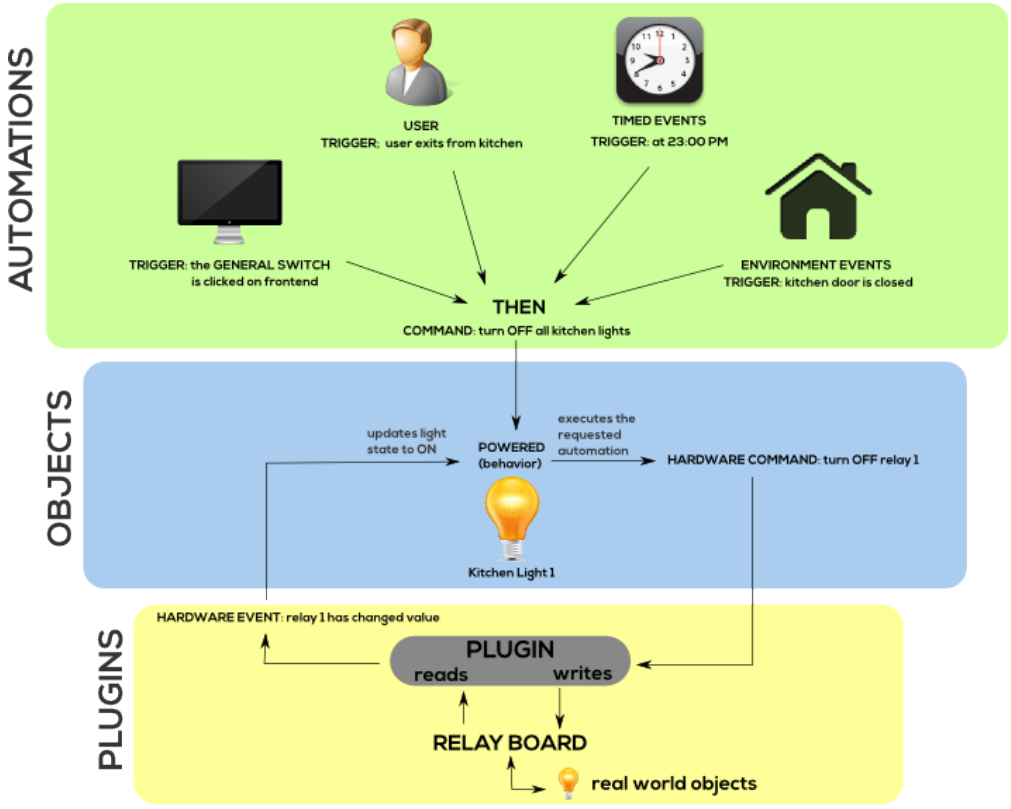


Fig. 9.2: Plugins, things and automations

9.3 Plugins, Objects and Automations interaction

The final goal is to define an automation which can **turn on** the living room light when it is tea time (17 o'Clock).

1. The scheduler plugin notifies to Freedomotic the current time (17:00 PM).
2. A trigger named "**it's tea time**" is configured to listen to all time based events. It carries a rule inside which is `event.time.hour == 17 AND event.time.minute == 0`.

3. When the event is received by this trigger, the rule is evaluated. If the evaluation is successful then the trigger fires, indicating that now it is not time to for tea.

#. At this point all the corresponding automation `IF (trigger: it's tea time) THEN (command: turn on livingroom light)` is loaded by the system and the command is executed which forwards the generic request `turn on living room light` to the plugin which can transform it to a protocol dependent command (eg: send string `A01AON` on serial port /dev/ttyUSB0).

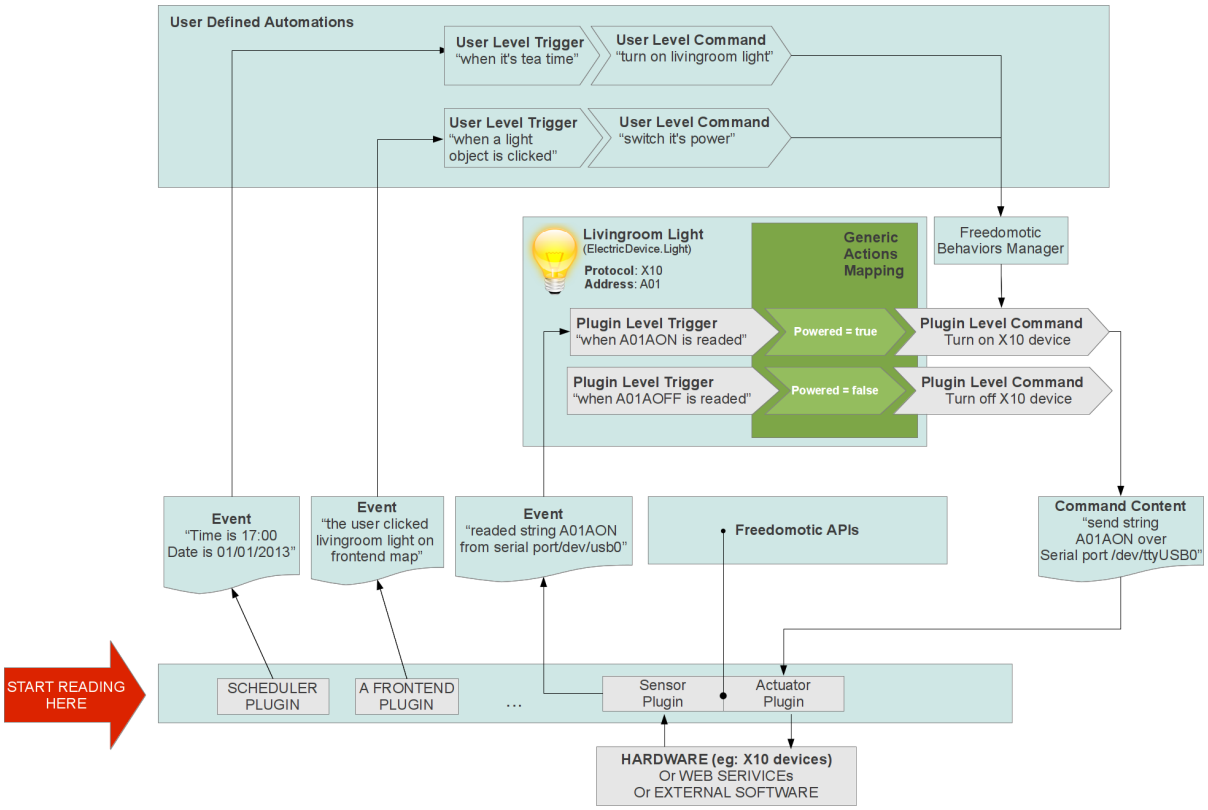


Fig. 9.3: Events, triggers, commands and automations

CHAPTER 10

Freedomotic Messaging System

Freedomotic uses simple structured messages (xml, json) to communicate with its components. This is done through a **Messaging Middleware** (**Apache ActiveMQ**).

Freedomotic is based entirely on events and any change in the environment and any user interaction (eg: a click on the GUI) generates events.

Events are published on **channels** and can be intercepted by **triggers**.

Each trigger may be associated with one or more commands defining a **reaction** or **automation**.

Utilizing this architecture, program behavior is not predetermined but is fully modificable at runtime, making it extremely flexible and adaptable to any possible use in building automation.

When a sensor communicates a change in the environment it sends out an event.

A trigger, which is a sort of an event filter, listens to the event subscribed to the channel on which this event is sent. If the event is consistent with the trigger one or more commands will be executed. The command is automatically sent to the actuator which is able to execute it.

- A sensor can be an hardware device like a luminosity sensor
- An event is fired by a sensor, for example `luminosity in the kitchen is 30%`
- A trigger can define an expression like `if luminosity is less than 50%`
- A command can be something like `turn on the light in the kitchen`
- An actuator can be relais board

The result of the interaction between event, trigger, and command is an automated action. In this case the automation is `if luminosity is less than 50% turn on the light in the kitchen`.

Note: Triggers and commands are defined by the user using the **Jfrontend** graphical **EventEditor**. Triggers, commands and automations are saved as XML files.

10.1 A Message Journey

The communication process of notification of an event to the execution of a command consists of several steps:

Event (notified by plugins or Freedomotic itself) -> **Trigger** (acting as an events filter to define simple use cases) -> **Command** (executed by an actuator)

In this example we will analyze an automation composed by a single command: `IF Livingroom light turns on THEN announce its status using text to speech`

What happens in the framework?

This is an event which describes a state change of a light which turns from OFF (powered=false) to ON (powered=true). This kind of events is notified on channel `app.event.sensor.object.behavior.change` by a sensor plugin for example a Modbus sensor.

Events can be notified through hardware protocol plugins, frontends or Freedomotic itself as in this case.

Here is an example event which informs all listeners that a 'thing' named **Livingroom Light** has changed:

```
<com.freedomotic.events.ObjectHasChangedBehavior>
  <eventName>ObjectHasChangedBehavior</eventName>
  <sender>Light</sender>
  <payload>
    <payload>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>date.dayname</attribute>
        <operand>EQUALS</operand>
        <value>Sunday</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>date.day</attribute>
        <operand>EQUALS</operand>
        <value>23</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>date.month</attribute>
        <operand>EQUALS</operand>
        <value>September</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>date.year</attribute>
        <operand>EQUALS</operand>
        <value>2012</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>time.hour</attribute>
        <operand>EQUALS</operand>
        <value>9</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>time.minute</attribute>
        <operand>EQUALS</operand>
        <value>45</value>
```

```
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>time.second</attribute>
        <operand>EQUALS</operand>
        <value>49</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>sender</attribute>
        <operand>EQUALS</operand>
        <value>Light</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>object.name</attribute>
        <operand>EQUALS</operand>
        <value>Livingroom Light</value>
      </it.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>powered</attribute>
        <operand>EQUALS</operand>
        <value>true</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>brightness</attribute>
        <operand>EQUALS</operand>
        <value>0</value>
      </com.freedomotic.reactions.Statement>
      <com.freedomotic.reactions.Statement>
        <logical>AND</logical>
        <attribute>object.currentRepresentation</attribute>
        <operand>EQUALS</operand>
        <value>0</value>
      </com.freedomotic.reactions.Statement>
    </payload>
  </payload>
  <isValid>true</isValid>
  <uid>116</uid>
  <executed>true</executed>
  <isExecutable>true</isExecutable>
  <creation>1348386349837</creation>
  <priority>0</priority>
</com.freedomotic.events.ObjectHasChangedBehavior>
```

You can define triggers to narrow any event just by listening on the event channel and setting a list of conditions (the statements) that must be met in order to consider this trigger as fired. The trigger can then be used as the "**WHEN/IF**" part of an automation (aka **scenario**).

Freedomotic starts with a set of predefined triggers which cover most use cases. At any time you can add new use cases using an existing trigger as a template.

```
<trigger>
  <name>Livingroom Light turns ON or OFF</name>
  <channel>app.event.sensor.object.behavior.change</channel>
  <payload>
```

```
    <payload>
      <statement>
        <logical>AND</logical>
        <attribute>object.name</attribute>
        <!-- allowed operand are EQUALS, REGEX, GREATER_THEN, GREATER_EQUAL_THEN,␣
↪LESS_THEN, LESS_EQUAL_THEN -->
        <operand>EQUALS</operand>
        <value>Livingroom Light</value>
      </statement>
      <statement>
        <logical>AND</logical>
        <attribute>powered</attribute>
        <operand>EQUALS</operand>
        <!-- here you can write true to select only 'turns on' cases -->
        <!-- here you can write false to select only 'turns off' cases -->
        <!-- ANY is used to match any case -->
        <value>ANY</value>
      </statement>
    </payload>
  </payload>
</trigger>
```

In an automation you bind a trigger to one or more commands. In this case the automation is `WHEN Livingroom Light turns on THEN Say electric device status`.

The command `Say electric device status` is shipped with the text to speech plugin (http://freedomotic.com/content/plugins/text-speech) and looks like this:

```
<command>
  <name>Say electric device status</name>
  <description>say electric device status</description>
  <receiver>app.actuators.media.tts.in</receiver>
  <properties>
    <properties>
      <property name="say" value="=
        if (@current.object.powered)
                say="@current.object.name is on with brightness at @current.object.
↪brightness";
        else
                say="@current.object.name is off";
          "/>
    </properties>
    <tuples/>
  </properties>
</command>
```

When a trigger is fired Freedomotic loads all related commands and evaluates them using runtime properties. So the command above will look like this when received by the **TTS Text to Speech** plugin.

Every plugin has access to time and date information, the set of properties defined in the event and the current object state if the event has something to do with environment objects (in this case a light).

Your plugin can use all this information for token substitution and scripting as for the 'say' property in the command above. In the command below you can see how the 'say' property is evaluated by Freedomotic before sending it to the text to speech plugin:

```
<command>
  <name>Say electric device status [EVALUATED]</name>
```

```
  <description>say electric device status</description>
  <receiver>app.actuators.media.tts.in</receiver>
  <properties>
    <properties>
      <!-- Static properties for the text to speech actuator. -->
      <!-- This are defined in data/cmd folder of the actuator itself -->
      <!-- The 'say' property is evaluated using runtime properties -->
      <property name="say" value="Livingroom Light is off."/>
      <!-- ALL PROPERTIES BELOW ARE EVALUATED AT RUNTIME -->
      <!-- generic data taken from the event which started the event-trigger-command␣
↪chain. -->
      <property name="event.sender" value="Light"/>
      <property name="event.date.dayname" value="Sunday"/>
      <property name="event.date.day" value="23"/>
      <property name="event.date.month" value="September"/>
      <property name="event.date.year" value="2012"/>
      <property name="event.time.hour" value="10"/>
      <property name="event.time.minute" value="30"/>
      <property name="event.time.second" value="24"/>
      <!-- the state of the object Livingroon Light when the event was fired -->
      <property name="event.object.name" value="Livingroom Light"/>
      <property name="event.brightness" value="0"/>
      <property name="event.powered" value="false"/>
      <property name="event.object.currentRepresentation" value="0"/>
      <!-- the current state of the object Livingroom Light (when this command is␣
↪executed -->
      <property name="current.object.name" value="Livingroom Light"/>
      <property name="current.object.type" value="EnvObject.ElectricDevice.Light"/>
      <property name="current.object.protocol" value="unknown"/>
      <property name="current.object.address" value="unknown"/>
      <property name="current.object.brightness" value="0"/>
      <property name="current.object.powered" value="false"/>
    </properties>
  </properties>
</command>
```

CHAPTER 11

Channels

The concept of **channel** is central to the messaging system as events and commands are published on specific channels.

Events are initiated by a sensor plugin. From Freedomotic's point of view a sensor is composed of a hardware device and software connected to the middleware that manages it.

Events can be exchanged in any of the supported formats (e.g. POJO, JSON, XML) and are communicated to the triggers through a **publish/subscribe** messaging channel. Each trigger must be subscribed to a channel to receive the events traveling through it.

11.1 Wildcard subscription

It is possible to use wildcards for subscriptions in order to automatically include an entire range of events. For example, if a sensor generates events on channel `app.events.sensors.moving.person.P003` a trigger can listen to this particular event to receive details about person P003's movements. Otherwise if the trigger listens to `app.events.sensors.moving.person.` it will receive information about the movement of all people detected in the environment.

The wildcard semantic is as follows:

- **period (.)** is used to separate names in a path
- **asterisk (*)** is used to match any name in a path
- **greater than sign (>)** is used to recursively match any destination starting from this name

A **trigger** is a filter that can be used to decide whether a notified event has to be processed or not. Whenever an event is processed by a trigger, the associated reaction is executed.

A **reaction** represents a link between a trigger and one or more command list executed by an actuator or another sensing system.

It allows to control the processing flow of the commands, capturing the resulting values of their execution.

Every command list is executed in parallel within a **dedicated thread** (therefore several reactions can occur in parallel). Each thread sequentially dispatches each command found in its own list using the **request/reply** pattern.

Notice that trigger and commands are reusable components, as they are not defined inside the reaction (which only specify the structured the execution flow).

A command identifies an outgoing message from the middleware containing the definition of the receiver specified as a Channel address (eg: `app.plugins.protocols.modbus.in`).

A command also contains all parameters needed to perform its execution. Commands are forwarded using request/reply messaging pattern while events use send-and-forget pattern. An actuator is the endpoint of the communication process.

An actuator can physically execute the command in the environment (e.g. turn a light on or open a window). It is also possible to query an actuator as if it were a sensor but only within its domain of control, for example it can reply to a query notifying the state of a light under its control.

Trigger, reactions and commands are defined in XML files that compose the model of all the world entities Freedomotic interacts with.

Such files are automatically loaded and saved to file by the middleware, ensuring data consistency.

11.2 Channel Examples

CHAPTER 12

Data structures

If you plan to develop a plugin for Freedomotic you will need to access the framework data structures.

These types of data are used with building automation and the Freedomotic architecture:

1. Environment topology
2. Environment objects
3. People
4. Loaded plugins and extensions

12.1 Environment topology

Environment data are accessible by the static reference. It returns the Environment's instance which gives you access to all of the Environment's properties

```
Freedomotic.environment.getPojo();
```

12.1.1 Returns all the zones in the environment

```
Freedomotic.environment.getZones()
```

Zones are logical (virtual) portions of the environment. To retrieve the list of physical environment rooms (rooms are also considered Zones) use:

```
Freedomotic.environment.getRooms()
```

12.1.2 Environmental 'Things'

The 'things' (lights, doors, couches, ...) in the environment can be retrived in different ways:

12.1.3 Get a 'thing' by its name

```
EnvObjectPersistence.getObject(String name)
```

12.1.4 Get the 'things' filtered by protocol and address property

```
EnvObjectPersistence.getObject(String protocol, String address)
```

12.1.5 Get the list of the 'things' linked to a specific protocol

```
EnvObjectPersistence.getObjectByProtocol(String protocol)
```

12.1.6 Get the list of all 'things' in the current environment

```
EnvObjectPersistence.getObjects()
```

Use the following import to access this method:

```
import com.freedomotic.objects.EnvObjectPersistence;
```

12.2 Plugins

12.2.1 Gets the list of loaded plugins

```
AddonManager.getLoadedPlugins()
```

Returns an ArrayList of Plugin type.

12.2.2 Get a plugin by name

```
AddonManager.getPluginByName(String name)
```

Remember to import com.freedomotic.plugins.AddonManager;

12.2.3 Get plugin configuration from manifest

You can access configuration file of a plugin in this way:

```
int myVar = configuration.getIntProperty("PROPERTY-NAME", 1);
```

The second parameter in getIntProperty is the default value to use if the *PROPERTY-NAME* cannot be found or cannot be converted to the proper type (int, double, string, ...)

other methods are:

```
boolean myVar = configuration.getBooleanProperty("PROPERTY-NAME", true);
double myVar = configuration.getDoubleProperty("PROPERTY-NAME", 1.5f);
String myVar = configuration.getStringProperty("PROPERTY-NAME", "some text");
```

read tuple properties from config file:

```
boolean myVar = tuple.getBooleanProperty(tupleIndex, "PROPERTY-NAME", true);
double myVar = tuple.getDoubleProperty(tupleIndex, "PROPERTY-NAME", 1.5f);
String myVar = tuple.getStringProperty(tupleIndex, "PROPERTY-NAME", "some text");
```

12.2.4 Get received command parameters

The onMessage method has a *Command c* parameter. Is possible to access the received parameters this way:

```
String saveItInAVariable = c.getProperty("COMMAND-PARAM-NAME");
```

12.3 Accessing Data Structures from Crosslanguage Plugins

This is done through a REST connection which serves the data. More info can be found at https://github.com/freedomotic/freedomotic/wiki/Freedomotic-APIs.

CHAPTER 13

What is a plugin?

Freedomotic is an application extensible through plugins. Plugins are simple classes within a .jar java package. Each plugin is deployed in the FREEDOMOTIC_ROOT/plugins/ folder, loaded, and initialized automatically at Freedomotic startup. The communication between the plugin and Freedomotic is automatically managed, via a Message Oriented Middleware. Plugins, in addition to the 'Manager of the messages', have direct access to Freedomotic data structures. In a plugin, you can create, read, update, or delete data and use them to accomplish your goals.

13.1 Plugin Features

1. Plugin configuration management
2. User interface accessible by right click in plugins list
3. Simplyfied access to freedomotic data structures
4. Automatic management of plugin lifecycle (loaded, running, stopped,...)
5. Access to the messaging system (read/write events and commands)
6. Installation and upgrade of plugins from a marketplace
7. Simplified programming implementing events (onCommand(), onRun(), onStart, onStop(), ...).

13.2 How to make a non-Java application communicate with Freedomotic

Till now we talked about how to extend Freedomotic with Java plugins. However is possible to make non-Java application communicate with Freedomotic. Take a look at https://github.com/freedomotic/freedomotic/wiki/Freedomotic-APIs

CHAPTER 14

Plugins manifest and configuration

Every plugin needs a XML manifest file to describe its configuration. As multiple plugins can be in the same package (the one you download from the marketplace) then more than one manifest file can be in the root folder of a plugin package.

The binding is made in the class constructor with

```
public MyPlugin() {
  super("Hello World Sensor" ,"/firstsensor/first-sensor-manifest.xml");
}
```

In this example the manifest file is *first-sensor-manifest.xml* located into *firstsensor* folder.

The path is case sensitive so */firstsensor/first-sensor-manifest.xml* is not the same as */FirstSensor/First-Sensor-Manifest.xml"*.

14.1 What's inside the manifest

This is an example of the most simple manifest file you can have:

```
<config>
    <properties>
       <property name="name" value="Hello World"/>
       <property name="description" value="A basic plugin that prints 'Hello World'␣
↪on standard output"/>
       <property name="category" value="examples"/>
       <property name="short-name" value="hello-world"/>
   </properties>
</config>
```

Note: The manifest file is the **ONLY** place where you should add configuration parameters for your plugin. You should not use external files if not strictly needed (e.g. by third party libraries).

You can add custom properties to this list. The properties can be retrieved programmatically this way:

```
int ServerPort = configuration.getIntProperty("udp-server-port", 7331); //defaults to␣
↪7331 if the property is not found in the manifest
String Delimiter = configuration.getStringProperty("delimiter", ":"); //defaults to ':
↪' if the property in not found in the manifest
```

14.2 Add configuration blocks to your plugin

If you have to configure multiple the same set of properties for different things (eg: URL and port of a set of hardware boards) you can use **tuples**.

You can add as many `<tuple></tuple>` blocks as you need. The `<tuples></tuples>` block may be added after `<properties></properties>` on the same hierarchical level.

Tuples are useful to have configuration data specific for you plugin to be loaded in Freedomotic at startup.

```
<tuples>
   <tuple>
     <property name="Name" value="TemperatureZone1"/>
     <property name="SlaveId" value="1"/>
     <property name="RegisterRange" value="HOLDING_REGISTER"/>
     <property name="DataType" value="TWO_BYTE_INT_UNSIGNED"/>
     <property name="Offset" value="266"/>
     <property name="NumberOfRegisters" value="1"/>
     <property name="Multiplier" value="0.1d"/>
     <property name="Additive" value="0.0d"/>
     <property name="EventName" value="TemperatureZone1"/>
  </tuple>
  <tuple>
     <property name="Name" value="TemperatureZone2"/>
     <property name="SlaveId" value="1"/>
     <property name="RegisterRange" value="HOLDING_REGISTER"/>
     <property name="DataType" value="TWO_BYTE_INT_UNSIGNED"/>
     <property name="Offset" value="522"/>
     <property name="NumberOfRegisters" value="1"/>
     <property name="Multiplier" value="0.1d"/>
     <property name="Additive" value="0.0d"/>
     <property name="EventName" value="TemperatureZone2"/>
  </tuple>
</tuples>
```

You can use free custom strings for the attribute name and the value.

14.3 Messaging channel

Every plugin has an unique input **messaging channel** used for message exchange. It is addressed using the info you put in the manifest file.

For plugins the channel name has the following format `app.actuators.CATEGORY.SHORT-NAME.in`.

CHAPTER 15

Create a new plugin

15.1 From template

1. Copy and paste the hello-world example you can find in GIT_FOLDER/plugins/devices/hello-world. Open this project along with **freedomotic-core** in your favourite IDE and make your changes. Any time you compile the plugin will be installed into *freedomotic-core* folder, so you can simply start it to see your plugin running.
2. Implement the **HelloWorld.java** class methods and rename the class and the path to manifest in the class Constructor according to the new name of your plugin.
3. Compile the plugin and start Freedomotic to test it.
4. Add commands, triggers and resources in the *src/main/resources/data/* folder of this plugin (take a look at the folder diagram above)

15.2 From an archetype

Archetypes can help you to make the development of new plugins easier starting from a well-defined template. More info on http://maven.apache.org/guides/introduction/introduction-to-archetypes.html

The archetype should be compiled with

```
mvn clean install
```

and added to the local list of archetypes with

```
mvn archetype:crawl
```

Now localhost is ready to create projects using this archetype

```
cd freedomotic/plugins/devices
mvn archetype:generate -Dfilter=com.freedomotic:device -DgroupId=com.freedomotic -
↪Dversion=1.0-SNAPSHOT -Dpackage=com.freedomotic
```

Add your plugin name when asked and confirm. The plugin skeleton is now created so compile it with:

```
cd PLUGIN_NAME
mvn clean install
```

At this point the new plugin is installed in Freedomotic as usual, the developer can start to code it.

Please remember to rename the default Java class **HelloWorld** as your plugin name.

15.3 Behind the scene

If you want to understand how the archetype works take a look here.

TODO Update maven archetype code to match latest freedomotic version and best practices

15.4 Plugin folder structure

```
frontend-java/
- pom.xml                                    //THE MAVEN POM FILE
- src
    - main
    |   - java
    |   |   - com
    |   |       - freedomotic
    |   |           - jfrontend
    |   |               - JavaDesktopFrontend.java    //THE PLUGIN SOURCE CODE
    |   - resources
    |       - data (PLUGIN DATA FOLDER)
    |       |   - i18n
    |       |       - jfrontend_it_IT.properties  //ITALIAN TRANSLATION
    |       |       - jfrontend.properties        //DEFAULT LANGUAGE
    |       |   - cmd                             //PLUGINS COMMANDS XML FILES
    |       |   - rea                 //PLUGINS REACTIONS/AUTOMATIONS XML FILES
    |       |   - templates           //OBJECTS TEMPLATES PROVIDED BY THIS PLUGIN
    |       |   - trg                 //PLUGINS TRIGGERS XML FILES
    |       |   - resources           //PLUGINS MEDIA FILES
    |       - desktop-frontend.xml    //THE PLUGIN MANIFEST
    - test
        - java
            - com
                - freedomotic
                    - jfrontend
                        - JavaDesktopFrontendTest.java //UNIT TESTS FILE
```

CHAPTER 16

Plugin lifecycle

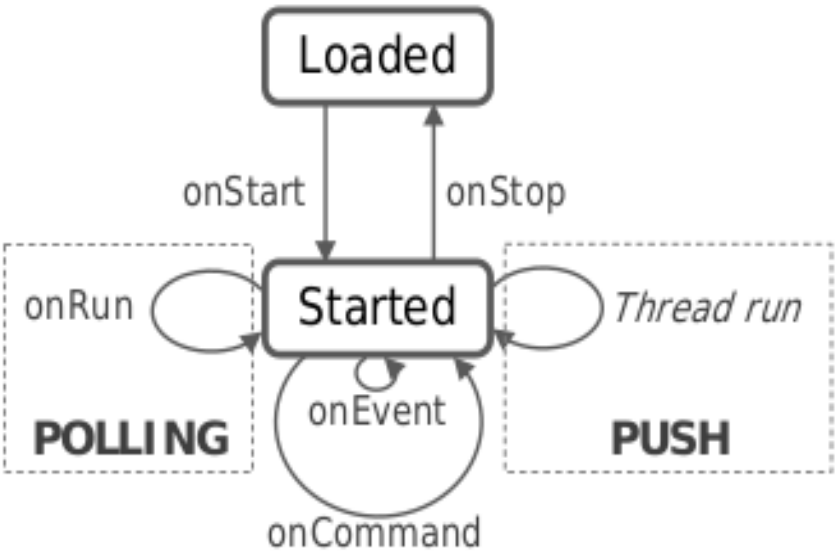


Fig. 16.1: Lifecycle of a plugin, showing both polling and push modes of operation (thanks to J.M. Fidalgo)

CHAPTER 17

Bind things state to hardware data

To integrate new building automation protocols, (eg: zwave, zigbee, ...) you can create a dedicated plugin.

This plugin acts as a "translator" from Freedomotic generic commands to protocol specific commands, and vice versa.

17.1 Read data from hardware

Your plugin will read data from the protocol and translate it into Freedomotic events to be notified to the framework. The event you want to notify is usually a ProtocolRead event; take a look at how to listen and notify events tutorial.

To bind an event to a change of the status of a specific object on the Freedomotic environment map, you have two ALTERNATIVE choices:

17.2 Specify the new object state in the notified event

High level communication protocols usually know if the read value is a temperature, a binary state (true/false) or another kind of value.

In this case you can specify the object state directly in the notified event as following:

```
//protocol name= zwave, address=3
ProtocolRead event = new ProtocolRead(this, "zwave", "3");
//set the object state to powered=true
event.getPayload().addStatement("behavior.name", "powered");
event.getPayload.addStatement("behaviorValue", "true");
// (OPTIONAL) specify a Freedomotic object class and name for the autodiscovery␣
↪feature
event.getPayload.addStatement("object.class", "Light");
event.getPayload.addStatement("object.name", "A Light");
//send the event to Freedomotic
notifyEvent(event);
```

The drawback is that your plugin is now bound to this specific object implementation, so it will work only with objects that have a behavior called **"powered"** which accepts true/false values (BooleanBehaviour class).

To avoid binding your plugin to a specific object implementation you can just notify the raw read hardware value and convert it into a valid behavior value using a hardware trigger (data source).

17.3 Create a hardware trigger to be configured as "Data source"

You would choose this option if your communication protocol doesn't know the object type. For example, a relay board just knows if a relay is **on** or **off** but not if the wired object is a **lamp** or a **coffee machine**. It simply notifies a hardware read value.

The interpretation of the raw read value is done at configuration time, because only the configurator knows what is really connected to the relay board. This simply means that to bind an object state (eg: `powered=true`) to a specific trigger, then you (as developer) must provide one along with your plugin.

For example if you are developing an Arduino based plugin you can define a data source trigger called `Arduino Relay Board: read value 1 in first relay line`. The mapping between the object state and your plugin trigger will make the object to become "powered" when the first relay of the Arduino board is switched on.

As a note, the object settings can be changed from JFrontend by right clicking on an object on the map (**Data source** tab).

17.4 Write data to hardware

Freedomotic can request something like this to your plugin `turn on relay 1 on board at 192.168.1.100`.

Assuming the board communication protocol is HTTP based, the plugin should translate this into a proper HTTP request to the IP the hardware board is listening.

Your plugin will receive this generic command in the `onCommand()` method of your plugin and here you would parse the command parameters with

```
c.getProperties().getIntProperty("propertyName", defaultValue)
```

and create the corresponding protocol specific request.

To know which variables are available to your plugin to perform its tasks take a look at the section Properties received by a driver plugin

CHAPTER 18

Bind things state to web services

18.1 Parse data from the Webservice

Here, you will probably need to request a specific URL content using a GET request. Then you will need to parse the received result, which is probably in JSON or XML. Take a look at http://stackoverflow.com/questions/4216455/get-page-content-from-url

All of this will be implemented in the `onRun()` method of your plugin.

This method is already threaded so there is no need to create additional execution threads. It can be executed just one time or in a loop with some delay between the calls. To do this, just call this method in your plugin constructor:

```
setPollingWait(2000);
```

This way the `onRun()` method is executed in a dedicated thread every 2 seconds. To disable the loop execution, just set it with a negative value:

```
setPollingWait(-1);
```

18.2 Send a Freedomotic event

We want it to the read temperature value for London using Weather Underground's APIs.

This event changes the temperature value of a thing on the map configured with protocol "**weather-underground**" and address "**london**". The temperature value is stored into a java variable "**londonTemperature**".

18.2.1 1. Specify the new state of the targeted thing in the notified event

```
//This value should be read from weather underground APIs in a real use case
int londonTemperature=21;
```

```
//Change the 'London Thermometer' object value
ProtocolRead event = new ProtocolRead(this, "weather-underground", "london");
//specify a freedomotic object type and name for the autodiscovery feature
event.addProperty("object.class", "Thermometer");
event.addProperty("object.name", "London Thermometer");
//set the 'temperature' behavior of 'LondonThermometer' object to 21
event.addProperty("behavior.name", "temperature");
event.addProperty("behaviorValue", londonTemperature);
//send the event to freedomotic
notifyEvent(event);
```

18.2.2 2. Create an event to be listened by triggers

If you just want to notify an event which is not directly related to the object's states, it can be done in the following way: Events are published by plugins on messaging channels. A series of useful events is predefined in Freedomotic but you can create your own or simply use the **GenericEvent** class.

Every action in the real environment and every interaction with Freedomotic (eg: a click on the GUI) is mapped to an event. Events can be intercepted by triggers, and you can assign one or more commands to a trigger, changing the behavior of the system at runtime.

18.2.3 How to notify a generic event

```
GenericEvent knowItAll = new GenericEvent(this);
//42 is the answer to the Ultimate Question of Life, the Universe, and Everything
//http://en.wikipedia.org/wiki/Answer_to_Life,_the_Universe,_and_Everything
event.addProperty("ultimate.question.answer", 42);
//set a channel on which this event should be sent
event.setDestination("app.event.sensor.deepthought");
//sends the event on the messaging bus
notifyEvent(event);
```

Now a trigger can listen to `app.event.sensor.deepthought` this way

```
<trigger>
    <name>You know the right answer to Life</name>
    <channel>app.event.sensor.deepthought</channel>
    <payload>
        <payload>
            <statement>
                <logical>AND</logical>
                <attribute>ultimate.question.answer</attribute>
                <operand>EQUAL</operand>
                <value>42</value>
            </statement>
        </payload>
    </payload>
</trigger>
```

and then you can create automations like `WHEN [You know the right answer to Life] THEN [send me an email]`

Besides the all purpose **GenericEvent**, some useful events are predefined in freedomotic. Look at this list http://freedomotic.com/javadoc/it/freedomotic/events/package-frame.html

Note: If your plugin's main purpose is to change the state of an object on the map (eg: set thermometer object value to the value read from Google Weather) then you should follow option 1.

18.2.4 More information about triggers

A trigger can listen on an events channel and filter the event content. If your event notifies about the outdoor temperature, then you can have a trigger called `Outside is cold` which fires if `temperature is less than 10°C`. You should provide this trigger along with your plugin in its *data/trg* folder. To know more about triggers definition take a look at this page /content/triggers.

18.2.5 An example: Get weather underground temperature data

TODO

CHAPTER 19

Handle plugin errors

When a plugin throws an exception, the related end user messaging can be handled automatically, for example setting the plugin description accordingly and stopping the plugin itself.

Here is an example of correct exception handling in `onRun()`

```
@Override
  public void onStart() throws PluginStartupException {
     try {
          // This code may generate a SerialPortException if there are connection␣
↪problems
          serial = new SerialHelper(PORTNAME, BAUDRATE, DATABITS, STOPBITS, PARITY,␣
↪new SerialPortListener() {
             @Override
              public void onDataAvailable(String data) {
                LOG.info("MySensors received: " + data);
                sendChanges(data);
              }
          });

          serial.setChunkTerminator("\n");
      } catch (SerialPortException ex) {
          throw new PluginStartupException("Error while connecting to serial device",␣
↪ex);
      }
  }
```

- **onStart()** throws **PluginStartupException**
- **onRun()** throws **PluginRuntimeException**
- **onStop()** throws **PluginShutdownException**

After catching the exception Freedomotic will:

1. print the error message on the GUI
2. change the plugin description using the exception message

3. log the exception
4. stop the plugin if it is a PluginRuntimeException

CHAPTER 20

Listen to Events programmatically

First you have to add a listener for each channel you want to listen to. In the onStart() method of your plugin add for example:

```
addEventListener("app.event.sensor.object.behavior.change");
addEventListener("app.event.sensor.environment.zone.change");
addEventListener("app.event.sensor.plugin.change");
```

if you want intercept all events about changing in sensors behaviors, zones or plugins.

Then modify the onEvent(EventTemplate event) method

```
protected void onEvent(EventTemplate event) {
        if (event instanceof ObjectHasChangedBehavior) {
            // here what you want todo
        } else if (event instanceof ZoneHasChanged) {
            // here what you want todo
        }
    }
```

Here a complete example.

CHAPTER 21

Auto discover and auto configure things

This feature is used to allow hardware plugins to create things automatically and place them on the already configured environment map just like an **auto discovering** system. The things will be created and added to the loaded environment the first time their state changes.

21.1 How to enable auto discovering in your plugin

Suppose you have some bulbs connected to a relay board. The first time you turn on one of them, Freedomotic generates a new light object and adds it to the map.

In order to do this you only need to send a state change notification for a thing. If the thing doesn't exist(checking is done on **protocol+address** values in the event), it is generated, configured and placed on the frontend map.

All the following examples are based on an X10 plugin, demonstrating the change in state from OFF to ON of the x10 device with address A01. Remember to change the values according to your needs.

```
ProtocolRead event = new ProtocolRead(this, "X10", "A01");
event.addProperty("x10.function", "ON"); // this is plugin related; your plugin will␣
↪have different properties
event.addProperty("object.class", "Light");
event.addProperty("object.name", "My X10 Light");
event.addProperty("object.protocol", "X10");
event.addProperty("object.address", "A01");
notifyEvent(event);
```

More information about these properties:

Property	Example Value	Description
object.class	Light	The type of the thing that will be created. It must be a string containing a thing type as you see in the things list menu of java frontend (when you press F6)
object.name	My Light	The name of the new thing. If the name already exists, a numeric ID will be added at the end of the name. For example: My Light 1
object.protocol	ProtocolName	(OPTIONAL) The name of the protocol used to manage this thing (eg: ZWave)
object.address	1234	(OPTIONAL) The address string (it is protocol dependent)

Note: Omitting object.class and object.name properties makes the ProtocolRead event to be discarded if no such thing exists. If thing exists the state change described in the event is applied.

CHAPTER 22

Internationalization

Plugins usually let the user interact with Freedomotic, e.g. by showing dialog boxes, logging messages and so on. I'd be advisable to let plugin use the user's language in order to make them much usable and understendable. If you're a developer and want to add your plugin the capability to 'speak' different languages, just follow this simple guide to adapt your code.

22.1 Adding internationalization support

Freedomotic ships a mechanism for easily support localized strings and allows the developer to use a prebuild bag of general purpose strings. Moreover the developer could add custom messages on his/her own.

First of all we need to access to API so let's add the following code

```
private API api;
private I18n i18n;

api = getApi();
i18n = api.getI18n();
```

The static function to use in place of your 'unlocalized' string is `i18n.msg(_STRING_KEY_)`.

Here a quick example

```
// old code (non localized)
LOG.info("Hello");
// new code (localized)
LOG.info(i18n.msg("greeting"));
```

In the plugin manifest file you have to add the property `<property name="enable-i18n" value="true"/>`.

22.2 Behind the scenes - what happens when calling i18n.msg()?

Freedomotic reads some system config to automatically guess user locale it searches proper localization string inside `/i18n/<Freedomotic locale>.properties`.

If the current locale is not defined (translation doesn't exist) **en_UK** is used and *Freedomotic.properties* file is loaded.

22.3 Making custom plugin translations

If global translation strings aren't enough plugin developer could write custom strings and save them using the following path starting from plugin base folder

`/i18n/<package-name>.properties\` for **en_UK**

or

`/<plugin_package_name>/i18n/<package_last_part>-<locale>.properties`

e.g.

`/jfrontend/i18n/jfrontend.properties`

`/jfrontend/i18n/jfrontend-es_ES.properties`

22.4 Accessing custom plugin translations

Just pass the current object as a parameter to `i18n.msg()`

```
LOG.info("Plugin " + i18n.msg(this,"plug_name"));
```

22.5 Composing strings

Consider the following example: we want to translate "**save environment as**", "**save object as**", "**save room as**" and so on. The translation file looks like this

Key string	Default translation	it_IT localization
save_as	Save as	Salva come
environment	Environment	Ambiente

using a concatenation of strings `i18n.msg("save_as") + i18n.msg("environment");` doesn't work and it'll result in "**save as environment**" ...

We can then use basic Java string format like that

Key string	Default translation	it_IT localization
save_as	Save {0} as...	Salva {0} come...
environment	Environment	Ambiente

```
i18n.msg("save_as",new Object[]{i18n.msg(environment)});
```

So the second variable is given as replacement for placeholder **{0}**.

This applies to many placeholders, not only one.

CHAPTER 23

Plugin samples

Take a look at the source code of these plugins to get inspiration

- SNMP Communication A plugin to interact with SNMP (Simple Network Management Protocol) enabled devices
- FTDI Communication
- UDP Communication Sending UDP packages to Open Picus devices
- I2C Communication
- MQTT Communication A MQTT client

CHAPTER 24

What is a thing?

The base class representing a thing is **EnvObject**. This class describes the main characteristics of a thing in the environment: identification, representations, behaviors and the ways it can interact with the environment (other things or users).

Things are stored in the application through the serialization of this class.

The class **EnvObjectLogic** incorporates EnvObject as a field, providing several methods to perform operations on it. This class serves as a template for the creation of objects.

Object behaviors (the properties you want to monitor or control) are defined as implementations of the interface **BehaviorLogic**.

Initially, depending on a template, objects can be considered virtual or not (it they have an attachment to a data source - a plugin). In order to establish correspondence with physical devices, an object must be marked as not virtual and a protocol and an address must be defined.

The **protocol** just indicates the plugin being used, while the **address** provides data specific to the protocol (for example the IP address of a network device or a web service or a board port).

These classes already provide representations of things on user interfaces, even allowing interaction with them.

Interaction with physical objects or web services is accomplished using plugins that translate generic data/commands from Freedomotic to the specific counterpart used in those devices/services and vice versa.

CHAPTER 25

Create new thing types

To develop a new object type, you have to create a Java extension which describes the actions that the new objects can perform.

After that your object can be instantiated by writing a XML file that describes the value of an instance of the object model you have implemented in Java.

To create the **Java object model**, you have to list the object properties and the values it can take. This is done by adding predefined listeners (called **behaviors**) to your object.

For example, a light can be turned on, turned off, and dimmed. So it has a behavior called **powered** that can be **true** or **false**, and a behavior called **brightness** that can assume integer values from 0 to 100.

A behavior is an instance of predefined classes. For example, the behavior **powered** is an instance of BooleanBehavior.java while **brightness** is an instance of RangedIntBehavior.java.

A behavior listens to change requests of its values, parses the request (for example, a sensor notifies that a light brightness has changed) and performs the defined operation for this situation.

The design pattern underneath is the same as a Java listener used for a Swing button. An example can be more clear. This is the definition of the brightness property of a light object

```
//linking this property with the behavior defined in the XML
//it takes the max, min and step values from the object definition file.
   brightness = (RangedIntBehavior) getBehavior("brightness");
    brightness.addListener(new RangedIntBehaviorListener() {
         @Override
          public void onLowerBoundValue(Config params) {
          //here you can add the code to execute if the brightness changes to the
          //lowest value possible. Eg: brightness equals to zero means the
         //light must be turned off.
           turnPowerOff(params);
          }
         @Override
          public void onUpperBoundValue(Config params) {
          //here you can add the code to execute if the brightness changes to the
          //highest value possible. Eg: brightness equals 100 means the
```

```
          //light must be set to on and not dimmed.
          turnPowerOn(params);
          }
         @Override
         public void onRangeValue(int rangeValue, Config params) {
         //here you can add the code to execute if the brightness changes to
         //a value inside the min-max range. Eg: brightness equals 45 means
         //the object must change its brightness value to 45.
         setBrightness(rangeValue, params);
          }
   });
```

the setBrightness() method will look like this

```
public void setBrightness(int rangeValue, Config params) {
        //executes the developer level command associated with
        //'set brightness' action defined in the object definition file.
        //the parameter 'params' has the data for the correct execution of the action.
        boolean executed = execute("set brightness", params);
        if (executed) {
            powered.setValue(true); //if dimmed, the light is on
            brightness.setValue(rangeValue);
            //set the light graphical representation
            setView(1); //points to the second element in the XML views array, the
↪"light on" image.
            setChanged(true);
        }
    }
```

25.1 Predefined behaviors

Here is a list of ready to use behaviors to instantiate as object properties:

Behavior Name	Listenable values changes	Example of use
RangedIntBehavior	onLowerBoundValue; onUpperBoundValue; onMiddleValue	Can be used to model a property like volume of a TV object
BooleanBehavior	onTrue; onFalse	Can be used to model the muted behavior of a TV object

25.2 Load the object as a plugin

As for any plugin the code must be compiled and its jar file must be deployed in the *FREEDOMOTIC_ROOT/plugins/objects/OBJECT_NAME_FOLDER*. As Freedomotic starts up, it loads all the objects inside the *FREEDOMOTIC_ROOT/plugins/objects/* subfolders irrespective of their names. Objects don't require a XML configuration file.

25.3 Create instances of your new object type

When it receives a specific input, Freedomotic knows how this type of object will execute. You will have to provide one or more *.xobj* files that describe the instances of your object type. For example, you have the light definitions but

you you need to add to the environment a light called 'kitchen light' and another one called 'living room light'. This is done through .xobj definition.

TODO: explain how to create a xobj object

- **An example** of Java class for a light object
- An xobj instance
- For a more challenging object take a look at TV object
- and its *xobj* instance

25.3.1 How to create the XML object

TODO: add a general description

25.3.2 Common properties section

Field	Values	Description	Required
name	String	The name of the object	YES
description	String	A brief description of your object (up to 100 char)	YES
actAs		NOT YET IMPLEMENTED	NO
type	EnvObject.ElectricDevice.Light	Dot notation of the object hierarchy in Freedomotic. It is a free form string you can use to identify	YES
protocol	String	Depends on the controller protocol eg: X10, Modbus,... Refer to the controller guide. Can be changed from the frontend at runtime.	YES
phisycalAddress	String	Depends on the controller protocol eg: X10, Modbus,... Refer to the controller guide. Can be changed from the frontend at runtime.	YES

25.3.3 Behaviors section

In this section the object's behaviors are configured. Each behavior name must have the same name that is used inside the object code. To facilitate the object's configuration an object developer should expose all names that is using inside the code. The names are case sensitive.

25.4 Boolean behavior

Used to describe a property that can have only two values: true or false. For example, the property **powered** of an electric device such a light.

Field	Values	Description	Required
name	eg: powered, muted, ...	the name of the boolean behavior	YES
description	String	A string to describe the behavior purpose	NO
value	Boolean	The startup value of the behavior	YES
active	Boolean	This behavior is valid on startup? If in doubt use "true"	YES
priority		NOT YET IMPLEMENTED	NO

25.5 Ranged int behavior

A behavior used to model a property that can have a ranged set of integer values. For example, from zero to hundred or the volume property of a TV object.

Field	Values	Description	Required
name	eg: powered, muted, ...	The name of the boolean behavior	YES
description	String	A string to describe the behavior purpose	NO
value	Boolean	The startup value of the behavior	YES
max	Integer	The upper value that can be assumed. Eg: 100	YES
min	Integer	The lower value that can be assumed. Eg: 0	YES
step	Integer	The step used to go to the next or previous value from the current one.	YES
active	Boolean	This behavior is valid on startup? If in doubt use "true"	YES
priority		NOT YET IMPLEMENTED	NO

25.6 Exclusive multivalue behavior

This behavior represents an object feature that only takes values from a predefined list. For example, the input property of a TV object can only take values like INPUT1, INPUT2, SATELLITE, etc...

Field	Values	Description	Required
name	eg: powered, muted, ...	The name of the boolean behavior	YES
description	String	A string to describe the behavior purpose	NO
active	Boolean	This behavior is valid on startup? If in doubt use "true"	YES
priority		NOT YET IMPLEMENTED	NO
selected	Integer	The default selected item	YES
list	List	The list of items. Each of them has the format item_value	YES

25.7 Views section

Each view corresponds to a visual representation of the object that can be shown using the object code. The position of the view on the list corresponds to the same number that is used in the code.

Field	Values	Description
tangible	Boolean	The object is a physical object or not
intersecable	Boolean	A person or shape can intersect this object
width	Integer	The width of the object
height	Integer	The height of the object
x	Integer	Its x position starting from 0,0 (the upper left corner) of the environment
y	Integer	Its y position starting from 0,0 (the upper left corner) of the environment
rotation	Integer	The rotation using the upper left corner of the object as pivot point
fillcolor / red	Integer	The color that fills the geometrical shape of the object
fillcolor / green	Integer	The color that fills the geometrical shape of the object
fillcolor / blue	Integer	The color that fills the geometrical shape of the object
fillcolor / alpha	Integer	The color that fills the geometrical shape of the object
textColor / red	Integer	The color of the text that describes the object
textColor / green	Integer	The color of the text that describes the object
textColor / blue	Integer	The color of the text that describes the object
textColor / alpha	Integer	The color of the text that describes the object
borderColor / red	Integer	The color of the shape border
borderColor / green	Integer	The color of the shape border
borderColor / blue	Integer	The color of the shape border
borderColor / alpha	Integer	The color of the shape border
shape/npoints	Integer	Number of points used to describe the shape
shape/xpoints	Integer	Ordered list of x coordinates of the points
shape/ypoints	Integer	Ordered list of y coordinates of the points
icon	String	The name of the icon in the resource folder (path can be omitted)

25.8 Actions section

The actions represent the tasks that can be performed by an object. These actions must be associated with the hardware command that has to be executed when the action is launched. As with the behavior, the name of each action must match the ones used in the object code. Also, the command value should match the name of an existing command (normally a hardware command created by the hardware plugin developer).

Field	Values	Description
name	String	The name of the action already defined in the object code
value	String	The name of the command

CHAPTER 26

Add new thing templates

This feature is used to allow device plugins to create fully customized objects using auto discovery feature. The object can come with the right mapping in **Data sources/Actions** or you can even change its representation (eg: a light with a custom icon).

To test it

1. Create a *src/main/resources/data/templates* folder inside a device plugin (eg: essential)
2. Copy the *light.xobj* object from **base-things** plugin in this new folder
3. Change the name of this file to *mytemplate.xobj* and change xml name tag to `<name>My Test Template</name>`
4. Recompile **essential** plugin to have these changes installed in **freedomotic-core**
5. Run **freedomotic-core**.

In **Jfrontend** enter **Objects Edit mode** and you should see a new template in the list called **My Test Template** which is provided by a device plugin instead of an object plugin.

Try to add it to the map as usual and check if the new light turns on when clicked.

CHAPTER 27

Events

Freedomotic and its plugins send events when anything relevant happens.

Any event is sent on a **messaging channel**. A channel address is a simple string with a hierarchical structure like `app.sensors.event.object.behavior.changed`.

You can subscribe an event channel from a trigger which is a filter of events.

For example if your event is `an object changed state` you can filter it using a trigger `if a light in the kitchen changed its powered state`.

Freedomotic events have a set of standard properties plus a list of properties related to the specific event.

27.1 Generic event properties

The following properties are common to all events and can be intercepted and filtered by any trigger:

PROPERTY	POSSIBLE VALUES	DESCRIPTION
date.dayname	eg: Sunday	English name of the day in which the event is throwed
date.day	eg: 4 for Thursday	The day number in which the event is throwed
date.month	eg: October	The month name in which the event is throwed
date.year	eg: 2016	The year number in which the event is throwed
time.hour	eg: 0-23	The hour number in 24h format in which the event is throwed
time.minute	eg: 0-59	The minute of the current hour in which the event is throwed
time.second	eg: 0-59	The second of the current minute in which the event is throwed
sender		The name of the module that have generated the event

27.2 Predefined events

Here a list of predefined events:

EVENT	CHANNEL	DESCRIPTION
AccountEvent	app.event.sensor.account.change	Account status changed
GenericEvent	app.event.sensor	Generic event. Use ONLY if there is not a specific event
LocationEvent	app.event.sensor.person.movement.detected	Person position detected
LuminosityEvent	app.event.sensor.luminosity	Luminosity changed
MessageEvent	app.event.sensor.messages.MESSAGE_TYPE	Message notified
ObjectHasChanged-Behavior	app.event.sensor.object.behavior.change	Object behavior changed
ObjectReceiveClick	app.event.sensor.object.behavior.clicked	Object clicked
PersonEntersZone	app.event.sensor.person.zone.enter	Person enters a zone
PersonExitsZone	app.event.sensor.person.zone.exit	Person exits a zone
PluginHasChanged	app.event.sensor.plugin.change	Plugin status changed
ProtocolRead	app.event.sensor.protocol.read.PROTOCOL_NAME	Protocol read
ScheduledEvent	app.event.sensor.calendar.event.schedule	Time related event
TemperatureEvent	app.event.sensor.temperature	Temperature changed
ZoneHasChanged	app.event.sensor.environment.zone.change	Zone changed

27.3 More info in Javadoc

For event specific data please refer to the Javadocs of the event classes https://freedomotic.github.io/javadoc/freedomotic-core/com/freedomotic/events/package-summary.html

For example this is the list of properties available to a trigger that listen to **ObjectReceiveClick** events

- date.day.name EQUALS Thursday
- date.day EQUALS 4
- date.month.name EQUALS October
- date.month EQUALS 10
- date.year EQUALS 2012
- time.hour EQUALS 18
- time.minute EQUALS 15
- time.second EQUALS 15
- sender EQUALS UnknownSender
- click EQUALS SINGLE_CLICK
- object.type EQUALS EnvObject.ElectricDevice.Light
- object.name EQUALS Light one
- object.protocol EQUALS X10
- object.address EQUALS A01

CHAPTER 28

Triggers

A **trigger** is a filter that permits to intercept an **event** on a **channel**.

It performs check on the event carrying values and tags and assigning a meaningful and reusable name to this restriction.

For example, an event can be the notification of time for example 10 o'Clock; a trigger can respond to time events. Suppose if the time is between 7 and 13 o'clock, you can name this trigger `it's morning` and reuse it to perform **reactions** like `IF it's morning THEN turn off outdoor lights`.

Therefore a trigger can be used to decide whether a notified event has to be processed or not. Whenever an event is processed by a trigger, if the trigger is consistent with its definition, then the associated commands are executed.

A **reaction** represents a link between a trigger and one or more commands list executed by an actuator.

In this brief tutorial the manual creation of an XML trigger is explained, however the end user can define triggers using the included graphical editor, so there is no need to edit the XML manually.

28.1 How to filter events using triggers

Events can be intercepted using triggers. each event has a default channel on which it is notified.

To know which is the default channel of a particular event see listenable events page (TODO ADD A LINK). To capture the event you just create a trigger that is listening to the same channel. For example, the **PersonMoving** event is published on the channel `app.event.sensor.person.movement.moving`.

To intercept a person's movement you can define a trigger listening to channel `app.event.sensor.person.movement.moving`.

28.2 How to filter received event parameters

As said before a trigger is an event filter. It can read event parameters and filter them according to the rules defined in it. Every rule is called **statement**. A statement is composed of a **logical** value, an **attribute** name, an **operand** and a

value.

The action tag can be used to listen on the default channel of a specif event. You have to insert the complete path of the Java class that implements the event. Otherwise, you can specify the channel with a custom string inside the `<channel> </channel>` tag.

- **logical**: is used to concatenate a statement with the previous statement. The default value is *AND*, means that the following rule is in logical AND with the previous. At this time only the AND logical value is implemented.
- **attribute**: it's the name of the event property to filter. To know which properties are carried by an event, you have to refer to the event page.
- **operand**: can be *EQUALS*, *REGEX*, *GREATER_THAN*, *LESS_THAN*, *GREATER_EQUAL_THAN*, *LESS_EQUAL_THAN*, *BETWEEN_TIME*. It's used to relate the attribute with the value.
- **value**: can be a string or an integer value. You can use the *ANY* key to match any value.

28.3 Max execution limit and flood control

Every trigger has a **max-executions** parameter which defines how many times this trigger can fire. This counter is reset at Freedomotic startup. If the value is **-1** this trigger has no max executions limit.

Another property if **suspension-time** which defines for how many milliseconds this trigger is disabled after firing. The trigger cannot fire again until its suspension time is finished. Every trigger has a standard suspension time of 100ms which can be overwritten if needed setting a lower value.

28.4 Listening to Channels with wildcard paths

This feature is applicable only if you insert the custom channel path as a string in the `[code] [/code]` tag.

For example if a sensor generates events on channel `app.events.sensors.moving.person.P003` a trigger can listen to this particular event to receive details about person P003's movements.

Otherwise if the trigger listens to `app.events.sensors.moving.person.` it will receive information about the movement of all people detected in the environment.

The wildcard semantic is as follows:

- **period (.)** is used to separate names in a path
- **asterisk (*)** is used to match any name in a path
- **greater than sign (>)** is used to recursively match any destination starting from this name

28.5 Trigger scripting

TODO ADD EXAMPLES TAKEN FROM

https://github.com/freedomotic/freedomotic/blob/master/framework/freedomotic-core/src/test/java/com/freedomotic/core/ResolverTest.java

28.6 Deploy a trigger

Triggers are deployed in the *FREEDOMOTIC_ROOT/data/trg* folder. They are files with **.xtrg** extension and are loaded at Freedomotic startup.

In the console you can have a view of the loaded triggers and the channel on which they are listening.

28.7 Examples

28.7.1 Check if a thing name in the event is 'Kitchen Light'

```
<statement>
  <logical>AND</logical>
  <attribute>object.name</attribute>
  <operand>EQUALS</operand>
  <value>Kitchen Light</value>
</statement>
```

28.7.2 Check if the thing type in the event is an electric device

```
<statement>
  <logical>AND</logical>
  <attribute>object.type</attribute>
  <operand>REGEX</operand>
  <value>^EnvObject.ElectricDevice\.(.*)</value>
</statement>
```

28.7.3 Check if the temperature in the event is strictly greater than 20℃

```
<statement>
  <logical>AND</logical>
  <attribute>@event.temperature</attribute>
  <operand>GREATER_THAN</operand>
  <value>20</value>
</statement>
```

28.7.4 Check if the given time (format: HH:mm:ss) is between the specified time interval (format: HH:mm:ss-HH:mm:ss)

```
<statement>
  <logical>AND</logical>
  <attribute>time.current</attribute>
  <operand>TIME_BETWEEN</operand>
  <value>23:00:00-8:30:00</value>
</statement>
```

28.7.5 Check is someone exits from kitchen

```
<trigger>
    <name>Someone Exits from Kitchen</name>
    <description>When someone exits from kitchen area</description>
    <channel>app.event.person.zone</channel>
    <payload>
        <payload>
            <statement>
                <logical>AND</logical>
                <attribute>zone</attribute>
                <operand>EQUAL</operand>
                <value>Kitchen</value>
            </statement>
            <statement>
                <logical>AND</logical>
                <attribute>person</attribute>
                <operand>EQUAL</operand>
                <value>ANY</value>
            </statement>
            <statement>
                <logical>AND</logical>
                <attribute>action</attribute>
                <operand>EQUAL</operand>
                <value>exit</value>
            </statement>
        </payload>
    </payload>
    <delay>0</delay>
</trigger>
```

This trigger can filter **PersonExitZone** events. In that case the trigger fires only if the event is related to the kitchen zone and the person **ID** can be **ANY** (valid for **ANY** person). If the trigger is consistent with the event one or more commands will be executed.

28.7.6 A thing of type Electric Device is clicked

```
<trigger>
  <name>When an electric device is clicked</name>
  <description>When an electric device is clicked</description>
  <channel>app.event.sensor.object.behavior.clicked</channel>
  <payload>
    <payload>
      <statement>
        <logical>AND</logical>
        <attribute>object.type</attribute>
        <operand>REGEX</operand>
        <value>^EnvObject.ElectricDevice\.(.*)</value>
      </statement>
      <statement>
        <logical>AND</logical>
        <attribute>click</attribute>
        <operand>EQUALS</operand>
        <value>SINGLE_CLICK</value>
      </statement>
    </payload>
```

```
  </payload>
  <persistence>true</persistence>
</trigger>
```

This trigger will listen (and filter) events of types **ObjectReceiveClick** because they are sent on channel `app.event.sensor.object.behavior.clicked`

These are the received parameters that can be used in the trigger above

- date.day.name EQUALS Thursday
- date.day EQUALS 4
- date.month.name EQUALS October
- date.month EQUALS 10
- date.year EQUALS 2012
- time.hour EQUALS 18
- time.minute EQUALS 15
- time.second EQUALS 15
- object.type EQUALS EnvObject.ElectricDevice.Light
- object.name EQUALS Light one
- object.protocol EQUALS X10
- object.address EQUALS A01
- sender EQUALS JavaFrontend
- click EQUALS SINGLE_CLICK

CHAPTER 29

Commands

When you create a new command, you can choose from two different ways. The first is the creation of an xml file deployed in the *FREEDOMOTIC_ROOT/data/cmd* folder. The second option is to use the **EventEditor** plugin.

The first choice is the best for developers because it guarantees full control of the values. This is because the **EventEditor** is still under development.

A Freedomotic command is a container of customizable parameters in the form of `parameter = value`.

Standard parameters to guarantee the correct routing of the message to the actuator that can execute the task are **name**, **description** and **receiver**.

Command xml files are messages used to instruct the actuators on which action must be performed.

Some commands are created at runtime, the same way the sensors create events. However, commands can be created at "design" time by the developer to have this command embedded in the plugin (in *PLUGIN_NAME/data/cmd* folder) or they can be created at runtime by the user/configurator (and it will be saved in *FREEDOMOTIC_ROOT/data/cmd* folder).

29.1 Commands fields

29.1.1 Properties received by a driver plugin

- @owner.*: all thing properties and behaviors with the value they had before automation execution. If an automation rises the light brightness values, the property `@owner.object.behavior.brightness` contains the starting value not the target value. **This is received only by driver plugins.**
- +Any plugin specific property, defined in the xml command into *data/cmd* folder of the plugin itself.

29.1.2 An example (turn on X10 device)

These are the properties received by a driver plugin which implements communication with X10 hardware.

- owner.object.protocol=X10

- owner.object.address=A01
- owner.object.name=Light one
- owner.object.type=EnvObject.ElectricDevice.Light
- owner.object.behavior.brightness=100
- owner.object.behavior.powered=false
- x10.gateway=PMIX35
- x10.function=ON

29.1.3 Properties received by a service plugin

- @event.*: contains all events properties
- @current.*: contains the properties of the event after the evaluation of the previous commands of this automation. If an automation rises the light brightness value, the property @current.object.behavior.brightness **contains the target value not the starting value**.
- +Any plugin specific property, defined in the xml command in data/cmd folder of the plugin itself.

29.1.4 An example (say ElectricDevice current state)

These are the command properties received by a text to speech plugin when the automation `IF a light turns on THEN say ElectricDevice current state` is performed.

- event.sender=Light
- event.date.day=4
- event.date.day.name=Thursday
- event.date.month=10
- event.date.month.name=October
- event.date.year=2012
- event.time.hour=18
- event.time.minute=15
- event.time.second=15
- event.object.type=EnvObject.ElectricDevice.Light
- event.object.currentRepresentation=1
- event.object.name=Light one
- event.object.protocol=X10
- event.object.address=A01
- event.object.behavior.powered=true
- event.object.behavior.brightness=100
- current.object.name=Light one
- current.object.type=EnvObject.ElectricDevice.Light
- current.object.protocol=X10

- current.object.address=A01
- current.object.behavior.powered=true
- current.object.behavior.brightness=100
- say=Light one is on with brightness at 100%.

29.2 The structure of a command

Field| Description ——-|————— name | A short string that identifies the effect of the command execution (eg: turn on light in the kitchen) description | An extended description of the effect of the command execution. Write it in the form "IF an event occurs THEN the system ... yourDescription receiver | The Channel on which the target plugin is listening to delay | Not Yet Implemented, let this parameter to 0 timeout | waiting time for the plugin reply, if 0 it's set to 10 seconds by default properties | A set of properties in form "key = value".

29.3 Commands for the BehaviorManager

These commands can be used to change objects state in `IF this THEN that` automations like `IF it's dark THEN turn on garden lights`.

Here some example of commands sent to the Freedomotic internal **BehaviorsManager**. It accepts a predefined set of properties keys but any plugin can have its own set.

29.4 Command examples

29.4.1 Turn on object named "livingroom light"

```
<command>
  <name>Turn on livingroom light</name>
  <receiver>app.events.sensors.behavior.request.objects</receiver>
  <description>turns on an object called livingroom light</description>
  <editable>true</editable>
  <properties>
    <properties>
      <property name="behavior" value="powered"/>
      <property name="value" value="true"/>
      <property name="object.name" value="Livingroom light"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

29.4.2 Switch power of all Light type things in all environments

```
<command>
  <name>switch power for all lights</name>
  <receiver>app.events.sensors.behavior.request.objects</receiver>
  <description>switch power for all lights</description>
  <editable>true</editable>
```

```
  <properties>
    <properties>
      <property name="behavior" value="powered"/>
      <property name="value" value="opposite"/>
      <property name="object.class" value="EnvObject.ElectricDevice.Light"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

29.4.3 Switch power of all Light type objects in room named 'Kitchen'

```
<command>
  <name>switch power for all kitchen lights</name>
  <receiver>app.events.sensors.behavior.request.objects</receiver>
  <description>switch power for all kitchen lights</description>
  <editable>true</editable>
  <properties>
    <properties>
      <property name="behavior" value="powered"/>
      <property name="value" value="opposite"/>
      <property name="object.class" value="EnvObject.ElectricDevice.Light"/>
      <property name="object.zone" value="Kitchen"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

29.4.4 Increase brightness (one step) of all Light type things in the environment

```
<command>
  <name>Increase lights brightness</name>
  <receiver>app.events.sensors.behavior.request.objects</receiver>
  <description>increases light brightness</description>
  <editable>true</editable>
  <properties>
    <properties>
      <property name="behavior" value="brightness"/>
      <property name="value" value="next"/>
      <property name="object.class" value="EnvObject.ElectricDevice.Light"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

29.4.5 Decrease brightness (one step) of all Light type things in the environment

```
<command>
  <name>Decrease lights brightness</name>
  <receiver>app.events.sensors.behavior.request.objects</receiver>
  <description>decreases lights brightness</description>
  <editable>true</editable>
```

```
  <properties>
    <properties>
      <property name="behavior" value="brightness"/>
      <property name="value" value="previous"/>
      <property name="object.class" value="EnvObject.ElectricDevice.Light"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

29.5 Command Scripting

Commands parameters can be scripted using javascript syntax like this:

```
<command>
  <name>Say the current temperature converted in Fahrenheit</name>
  <receiver>app.actuators.media.tts.in</receiver>
  <delay>0</delay>
  <timeout>2000</timeout>
  <description>say the current temperature using TTS engine</description>
  <hardwareLevel>false</hardwareLevel>
  <persistence>true</persistence>
  <executed>false</executed>
  <properties>
    <properties>
      <property name="say" value="= say="The current temperature in @event.zone is "␣
↪+ Math.round(((@event.temperature+40)*1.8)-40) + " Fahrenheit degrees. In Celsius␣
↪is @event.temperature degrees"/>
    </properties>
    <tuples/>
  </properties>
</command>
```

This command uses text to speech to say the current temperature in a zone and makes a on the fly conversion from degrees Celsius to degrees Fahrenheit. The property key is a variable in the scripting context that can be evaluated.

To make a value scriptable it must start with an "**=**" just like Excel. Values that do not start with "**=**" are the same as the previous Freedomotic versions.

Here other example of scripting:

```
//sum the first 10 integer and store the value in myVar property
<property name="myVar" value="= myVar=0; for (i=0; i<10; i++) myVar+=i;"/>
```

```
//if one is one myVar property is one
<property name="myVar" value="= if (1==1)  myVar=1; else myVar="AREYOUJOKING?";"/>
```

```
negate the powered value of a thing if true becomes false, if false become true
<property name="myVar" value="= myVar=!@event.object.powered;"/>
```

CHAPTER 30

Reactions (aka Automations)

Reactions are based on the concept of Channel, so be sure to have understood this concept before you continue reading.

A **reaction** consists of a trigger and at least one or more commands. The listed commands are executed sequentially. The reactions run in parallel within a dedicated thread for each of them. The triggers and the commands are defined in files that independent from the same reaction, which represents only a link. So it is possible to reuse commands and triggers in different reactions.

Example:

- Reaction Name: entertainment scenario
- Trigger: TV turns ON
- Command Sequence 1: Turn OFF Livingroom lights
- Command Sequence 2: Close Windows -> Close Blinds

When it is Monday evening, and the TV turns ON, the lights in the livingroom are switched off. At the same time, the windows begin to close, when all the windows are completely shuted, the system begins to lower the blinds.

XML Representation Reaction are deployed in **FREEDOMOTIC_ROOT/data/rea** folder. This is the XML describing the previous scenery.

```
<reaction>
  <trigger>TV turns ON</trigger>
  <sequences>
    <sequence>
      <command>Turn OFF Livingroom lights</command>
    </sequence>
    <sequence>
      <command>Close Livingroom windows</command>
      <command>Close Livingroom blinds</command>
    </sequence>
  </sequences>
</reaction>
```

30.1 Composing triggers in automations (extra-conditions)

This means is possible to create automations like `IF [it's morning] AND [livingroom light is on] THEN [do something].`

The **extra conditions** feature is represented by `AND [livingroom light is on]` part which allows you to lookup for the current value of any object on the map to create additional conditions which are evaluated before your automation is executed.

There is still no frontend support for this feature, you should define it in XML editing the XML file in the **data/rea** folder (is the folder which contains the **automations**, AKA **reactions**).

Here an example `WHEN a door is clicked AND livingroom light is on OR the kitchen light is on THEN switch the open state of the clicked door`

```
<reaction>
  <trigger>When a door is clicked</trigger>
  <conditions>
    <condition>
        <target>Livingroom Light</target>
        <statement>
            <logical>AND</logical>
            <attribute>powered</attribute>
            <operand>EQUALS</operand>
            <value>true</value>
        </statement>
    </condition>
    <condition>
        <target>Kitchen Light</target>
        <statement>
            <logical>OR</logical>
            <attribute>powered</attribute>
            <operand>EQUALS</operand>
            <value>true</value>
        </statement>
    </condition>
  </conditions>
  <sequence>
    <command>Switch its open state</command>
    <command>test</command>
  </sequence>
</reaction>
```

TODO ADD NEW SYNTAX EXAMPLE FOR EXTRA CONDITIONS

CHAPTER 31

Http helper

This helper retrieves url content (html, json, xml) as string by performing http GET requests. It can also perform xpath queries on the retrieved content.

Let's start with an example. We want to retrieve the temperature in Rome from an online service. First of all create a new helper

```
HttpHelper http = new HttpHelper();
```

Remember that is a best practice to reuse this object if you have to do multiple requests

31.1 Retrieve text content (no authentication)

Using `retrieveContent(String url)` you can specify the online service url to retrieve data from and print the string on the console

```
String xml = http.retrieveContent("http://api.openweathermap.org/data/2.5/weather?
↪q=Roma&mode=xml");
System.out.println(xml);
```

Here the complete code included into a `<try><catch>` block to manage IO exceptions

```
try {
  String xml = http.retrieveContent("http://api.openweathermap.org/data/2.5/weather?
↪q=Roma&mode=xml");
  System.out.println(xml);
} catch (IOException ex) {
  //handle exception here
}
```

31.2 Retrieve text content (with authentication)

If the service requires authentication you can use `retrieveContent(String url, String username, String password)`.

31.3 Perform XPath queries on an URL content

In this example we are quering an XML Rest API to retrieve the current temperature in Rome and the related unit in which the temperature value is expressed. Note: Instead of null values you can pass username and password to access the service. **Note: XPath queries works on XML content only**

```
try {
  List<String> values = http.queryXml(
      "http://api.openweathermap.org/data/2.5/weather?q=Roma&mode=xml", null, null,
      "//current/temperature/@value",
      "//current/temperature/@unit");
  // Prints 'Temperature in Rome is 234 kelvin'
  System.out.println("Temperature in Rome is " + values.get(0) + " " + values.get(1));
} catch (IOException ex) {
  //handle exception here
}
```

CHAPTER 32

Serial helper

This service is based on JSSC library.

First of all you must create a new SerialHelper and set your port parameters in the following order:

- **port name** (*/dev/ttyUSBx* or */dev/ttyACMx* for Linux; *COMx* for Windows)
- **baud rate**
- **data bits**
- **parity bit**
- **stop bits**

Also you can override *onDataAvailable(String data)* method to define how to manage read data. For example you can send received data to another method.

```
SerialHelper usb = new SerialHelper("/dev/ttyUSB0", 19200, 8, 1, 0, new␣
↪SerialPortListener() {
  @Override
  public void onDataAvailable(String data) {
    System.out.println("DEBUG: received: " + data);
  }
});
```

By default data are read continuously. If you want to read a chunk of data with a specific terminator char you can set it as

```
usb.setReadTerminator("\n");
```

or if you want to read a chunk with a fixed dimension you can set it as

```
usb.setChunkSize(5);
```

Sending a string to the serial port is very simple

```
usb.write("ABCD");
```

Here the complete example

```
SerialHelper usb = new SerialHelper("/dev/ttyUSB0", 19200, 8, 1, 0, new␣
↪SerialPortListener() {
  @Override
  public void onDataAvailable(String data) {
    System.out.println("DEBUG: received: " + data);
  }
});

usb.setReadTerminator("\n"); //receive until this terminator char
//ALTERNATIVE: usb.setChunkSize(5); //receive messages of 5 chars each
usb.write("ABCD");
```

32.1 Complete examples

Arduino USB plugin

CHAPTER 33

Udp helper

This helper allows to start an UDP server listenig on a specific port and send UDP packets to a remote host.

Let's start with an example. We want to add a server listening on port 7777.

First of all create a new helper and start the server specifying ip address, port number and a listener

```
UdpHelper udpserver = new UpdHelper();
udpServer.startServer("0.0.0.0", 7777, new UdpListener() {
          @Override
          public void onDataAvailable(String sourceAddress, Integer sourcePort,␣
↪String data) {
              System.out.println("UDP packet received: {0}", data);
              // here add the code to execute when a packet is received
          });
```

When a new packet arrives you can see the payload on the log as a string and execute any code added to **onDataAvailable** method.

It'd be better to stop the server and unbound the port with

```
udpServer.stop()
```

33.1 How to send a packet

To send a packet you need only the ip and port number of the remote host and a string representing the payload.

For example:

```
udpServer.send("192.168.0.120", 8899, "payload to send");
```

CHAPTER 34

Natural language processing

It takes in input some text, analyzes it to compute a similarity value related to a set of predefined objects available for the system (eg: automation commands)

34.1 How to work

It makes a text analysis and ranks objects according to their similarity value. These objects are typically Commands, for example a speech recognition algorithm returns a text and you want to identify the most similar predefined command to execute it. Beware that computing similarity may be CPU intensive. Similarity may be computed with different algorithms. Our solution is based on Damerau-Levenshtein distance. View our implementation.

34.2 How to use it

The software listens to free-form (natural language) text commands on channel `app.commands.interpreter.nlp` and executes most similar command that the framework has in memory. For example a speech recognition utility may return a free-form text that can be interpreted by this module as an executable command. Another example is a chat bot that executes text commands.

34.3 Code sample

```
Command nlpCommand = new Command();
nlpCommand.setName("Recognize text with NLP");
nlpCommand.setReceiver("app.commands.interpreter.nlp");
nlpCommand.setDescription("A free-form text command to be interpreded by an NLP module
↪");
nlpCommand.setProperty("text", text);
nlpCommand.setReplyTimeout(10000);
Command reply = send(nlpCommand);
```

A complete code sample can be found here.

CHAPTER 35

P2P

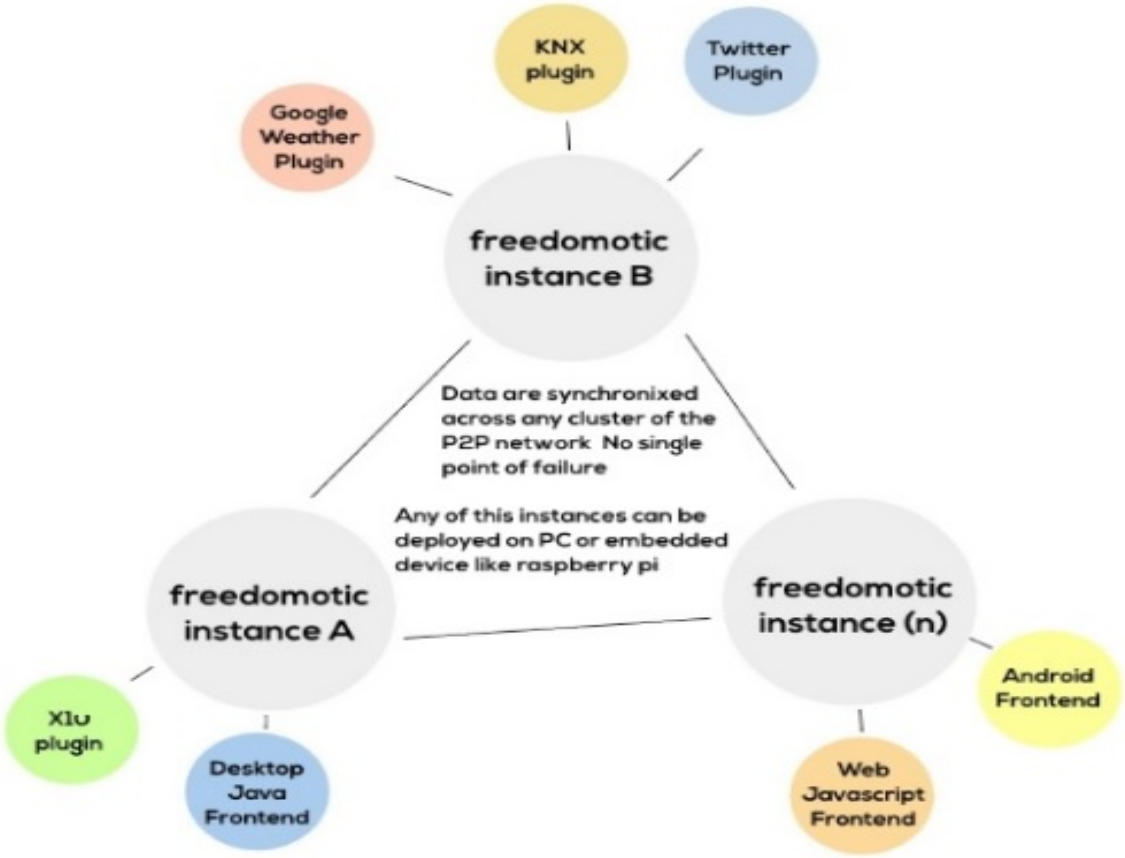
Implementation of p2p synchronization of different Freedomotic instances. If you turn on a light in instanceA the change is reflected in instanceB. Also moving an object changes its location in both instances.

In details

- there is no need to duplicate a plugin installing it on many instances, expecially for hardware related plugins where should be only one. The system is really distributed (despite the current data synch mechanism which is not perfect). If a plugin is duplicated you get load balance between the instances. As the core is always duplicated the core tasks are always load balanced between instances
- messaging pattern: the first command will be executed by instanceA, the second by instanceB and so on in a round robin way splitting the load.

Each instance has a complete copy of the data so you can turn off the other instances when you want. For sure you'll loose the "local not replicated plugins" but the system will be online and you will not notice any "recovery delay" because each instance always uses its local data, which are silently synchronized with the others.

Now we have a "state synchronization" but when we'll refine the mechanism we will have a "completely replicated cache" which is far better than the current system.



35.1 Roadmap

To reach this goal we made some changes:

- identified a single Freedomotic instance using an unique id and appended this id to each generated message
- changed the messaging broker url to **peer://freedomotic/INSTANCE_UUID**
- changed broker **websocket** and **stomp** endpoints to run on the first available port, otherwise it couldn't start two different Freedomotic instances on the same machine as the port would be already in use. This means the port for **ws** and **stomp** endpoints will change at each running.

35.2 TODO

- implement a "fix port number" in the **config.xml** file to make it static when needed
- add a SynchManager component (already developed in its first raw version)
- get startup data from another already running Freedmotic instance

35.3 How to use

Workaround for now is sharing the **data** folder on Dropbox and make both instances to read from the same folder to start. No problem if you run both instances from the same PC because they already share a common **data** folder.

Note: The restapi3 port is still static (9111) so the second Freedomotic instance will not be able to start this plugin. If you run two instances from the same system the problem is automatically solved (the second one cannot start because the port is already in use). If Freedomotic is installed on two RaspberryPis, then restapi plugin should be removed from the other instance before startup.

CHAPTER 36

Security: authentication and authorization

Freedomotic uses Apache Shiro, a Java security framework, to manage authentication and authorization.

This tool is very flexible, and offers many other features as cryptography and session management. Also, it's very easy to configure and use.

All the classes are accessible from this folder.

Currently we adopt Apache Shiro v1.3.2

36.1 Authentication

The class UserRealm makes the work. All users' data are stored in *users.xml* files located in *config* folder.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <users>
  <user>
   <principals>
     <principal realm="com.freedomotic.security" primary="true">system</principal>
   </principals>
   <credentials>p9aAW+vW4EWkTHsfNWWJcTGBOhUya1ORvu/dvU/A+0g=</credentials>
   <salt>zpGvSXleABptoH8/eq/xMrIkeStJzCT4JbNUe7LpL9g=</salt>
   <roles>
     <role name="administrators"/>
   </roles>
   <properties>
     <property name="language" value="auto"/>
   </properties>
  </user>
  <user>
   <principals>
     <principal realm="com.freedomotic.security" primary="true">admin</principal>
   </principals>
   <credentials>rfx9xwecAlh7Z45fUK+Gi+CSOirq1U2IdmQBTHXoeCw=</credentials>
   <salt>fX5WIcGU2ESqc6ECdOccJuM1ox5brXrOYxWw0EpJTHY=</salt>
```

```
   <roles>
     <role name="administrators"/>
   </roles>
   <properties>
     <property name="language" value="auto"/>
   </properties>
  </user>
  <user>
   <principals>
     <principal realm="com.freedomotic.security" primary="true">guest</principal>
   </principals>
   <credentials>TRN2QAmWB9oPk2E9JSZOcQxDmOZU/G1BGrjB92KQfPA=</credentials>
   <salt>vwoYXjSDQzqvr05h+xBprF5pCzGgQhfMAiG95kk67x4=</salt>
   <roles>
     <role name="guests"/>
   </roles>
   <properties>
     <property name="language" value="auto"/>
   </properties>
   </user>
 </users>
```

Credentials are saved in hash format (SHA-256).

Every user has a specific **role** as reported in *roles.xml* file.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <roles>
  <role name="administrators">
   <permissions>
     <permission>*</permission>
   </permissions>
  </role>
  <role name="system">
   <permissions>
     <permission>*</permission>
   </permissions>
  </role>
  <role name="guests">
   <permissions>
     <permission>*:read</permission>
   </permissions>
  </role>
  <role name="managers">
   <permissions>
     <permission>*:create,read,update,delete</permission>
   </permissions>
  </role>
</roles>
```

A **role** defines a system profile, and gives some permissions to interact with the system.

We have four different roles: **administrators**, **system**, **guests** and **managers**. The first two have unlimited privileges.

36.2 Authorization

Privileges are managed via *privileges.list* file. Each section reports a list of allowed actions.

```
#Currently supported and used privileges

[environments]
environments:create
environments:read
environments:update
environments:delete
environments:load #from file
environments:save #to file

[zones]
zones:create
zones:read
zones:update
zones:delete

[objects]
objects:create
objects:read
objects:update
objects:delete
objects:load #from file
objects:save #to file

[system]
sys:config:load
sys:plugins:load
sys:plugins:read
sys:plugins:start
sys:plugins:stop
sys:plugins:update
sys:shutdown

[auth]
auth:privileges:update
auth:privileges:read
auth:realms:create
auth:realms:delete

#Privileges to be added soon

[triggers]
[reactions]
[commands]
```

CHAPTER 37

Freedomotic API

Freedomotic exposes its functionalities by RestAPI v3 plugin.

It implements a RESTful service based on Jersey and Atmosphere framework.

So it's possible to create a custom client (not only in Java but in every language can consume a REST service) to interact with our platform.

For more details about the plugin configuration you can visit this link.