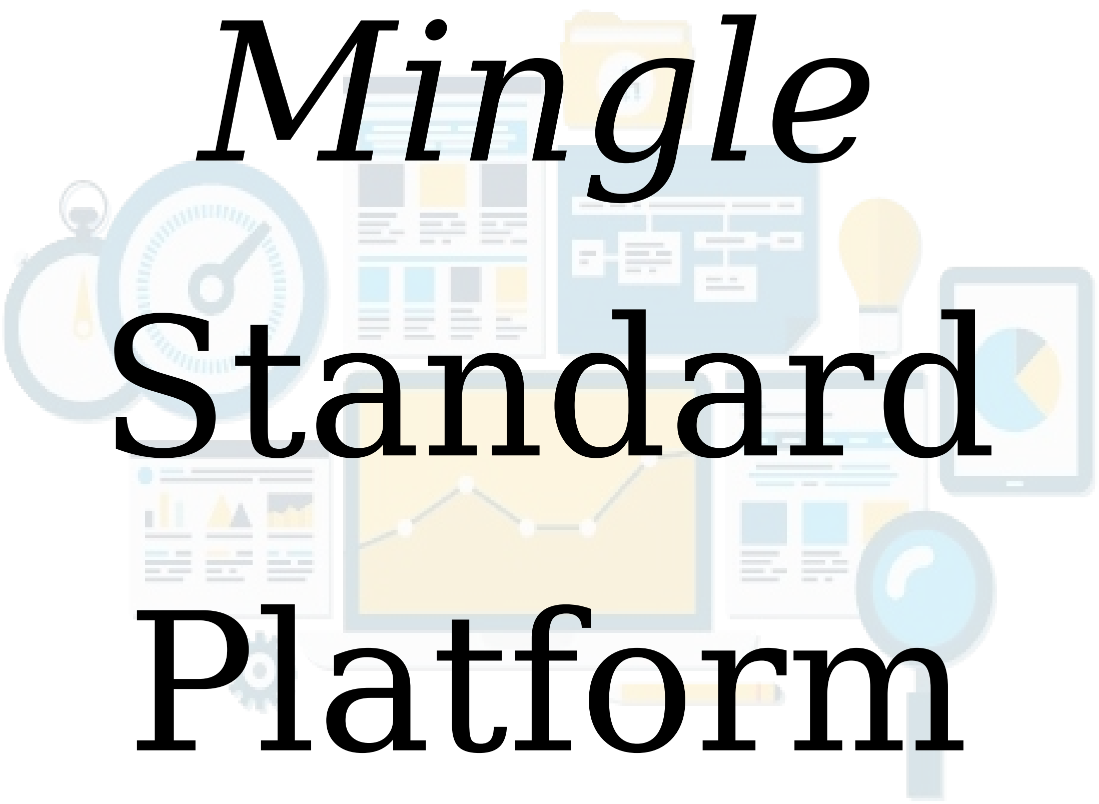


Mingle Standard Platform

Francisco Morero Peyrona

Preface

This work was initiated as per my PhD thesis.

It is dedicated to my family: their partners, children, parents and siblings (recursively).

About the mascot

The mascot for the *Une* language is an Electric Anguilla.

I bought the rights to use the image and I allow everyone to use it as far as it is related to the *Une* language or the Mingle Standard Platform.

For more information about this amazing animal, visit:
https://en.wikipedia.org/wiki/Electrophorus_electricus

About the names

- **Mingle**
  This is the name given to the whole ecosystem (the platform): the language and all the tools around it.

  mingle /mĭng′gəl/[1]

  1. To mix or bring together in combination.
  2. To be or become mixed or united.
  3. To associate or take part with others.

  Meaning that this platform aims to be the glue for the IoT. All following names, are referred in one or another way to the idea of “Glue”.

- **Une**
  This is the name of the language. In Spanish language it means “what joins”.

- **Tape**
  Glue can exist in the form of a tape: used mainly when wrapping gifts. This is the name of the transpiler (a special type of compiler). The transpiler transforms and optimizes source *Une* code into its abstract representation.

1 The American Heritage® Dictionary of the English Language, 5th Edition.

- **Stick**
  Glue-sticks are mainly used by kids. It is the name of the Execution Environment: it loads an *Une* code abstract representation and executes it.

- **Glue**
  This is the traditional form of: a viscose liquid. This is the name of the IDE (Integrated Development Environment).

- **Gum**
  This is an ancient form of glue. This is the name of the monitoring tool.

Special thanks

To (in alphabetical order):

- Gabarrón Luque, Anto.
- Martín Gómez, Juan Domingo

Summary

What is the Mingle Standard Platform?

It is a set of tools (software applications) where all components complement the others, forming an ecosystem whose goal is to make the “Internet of Things” (IoT) accessible to everyone.

Those with a basic understanding of spreadsheets (e.g., LibreOffice Calc, MS Excel, etc.) can learn and utilize Mingle in a single day. Those with a more advanced understanding of spreadsheets may be able to undertake more complex projects. Furthermore, software developers may approach projects of varying complexity with minimal investment of effort.

Mingle aims to be the “*lingua franca*” for the *Internet of the Things* (IoT) world. The main problem the IoT has nowadays is the incompatibility: every device has its own “language” and no one “speaks” with the rest of devices.

This is exactly what Mingle would like to solve. By providing a language that can be learn by almost anyone in few hours and by providing a set of standards that will allow every device to communicate with any other device that adheres the standard.

If you are familiar with Data Bases, you’ll understand all this perfectly with the following sentence: Mingle wants to be to the IoT field what SQL is to Data Bases field.

What the word “Standard” refers to?

In the end, Mingle is a set of software specifications that can be incarnated in many ways: different software companies can create different implementations of the specifications.

When we talk about the “Mingle Standard Platform”, we are mentioning the implementation that is used as the reference implementation. This is the one we will cover in this handbook, as the title of this book mentions. You can get the last version of the “Mingle Standard Platform” version here:

https://mingle.peyrona.com/

Mingle implementation was written using mainly Java language and was developed in such way that other developers can easily extend and replace certain modules by their owns. How this can be achieved, is out of the scope of this handbook, but lets say that developers will find a set of “Java interfaces” that will facilitate the understanding, recreation and reuse of the modules that comprises the “Mingle Standard Platform”.

As the title of this handbook suggests, all that is said here, refers to the *Mingle Standard Platform*; from now and on: **MSP**.

What is the MSP price?

Zero.

Mingle is Open Source (under the MIT license[2]) software and therefore can be used by anyone and all the code can be freely used and modified.

And the same about the documentation: it can be freely used, modified and extended by anyone.

You can get the last Mingle Standard Platform version here:
https://mingle .peyrona.com/

What is needed to run MSP?

All you need is a computer that can execute Java. Almost all computers can do it, and the MSP comes with the appropriate Java for Windows, Mac and Linux. If you are not using any of these 3 Operative Systems, all you need to do is to install the Java JDK for your OS and take a look to the “run.sh” file.

Almost any computer, even an old Raspberry Pi[3] (and others similar computers that cost about 35 US$) is more than enough to run all MSP tools.

Other implementations could have different requirements; e.g. not need Java.

Which components compose the Platform?

Well, it depends: as it was previously said, different implementations could have different tools, not only the total number of the available tools, but also in their scope, utility and purpose.

At least the following are needed:

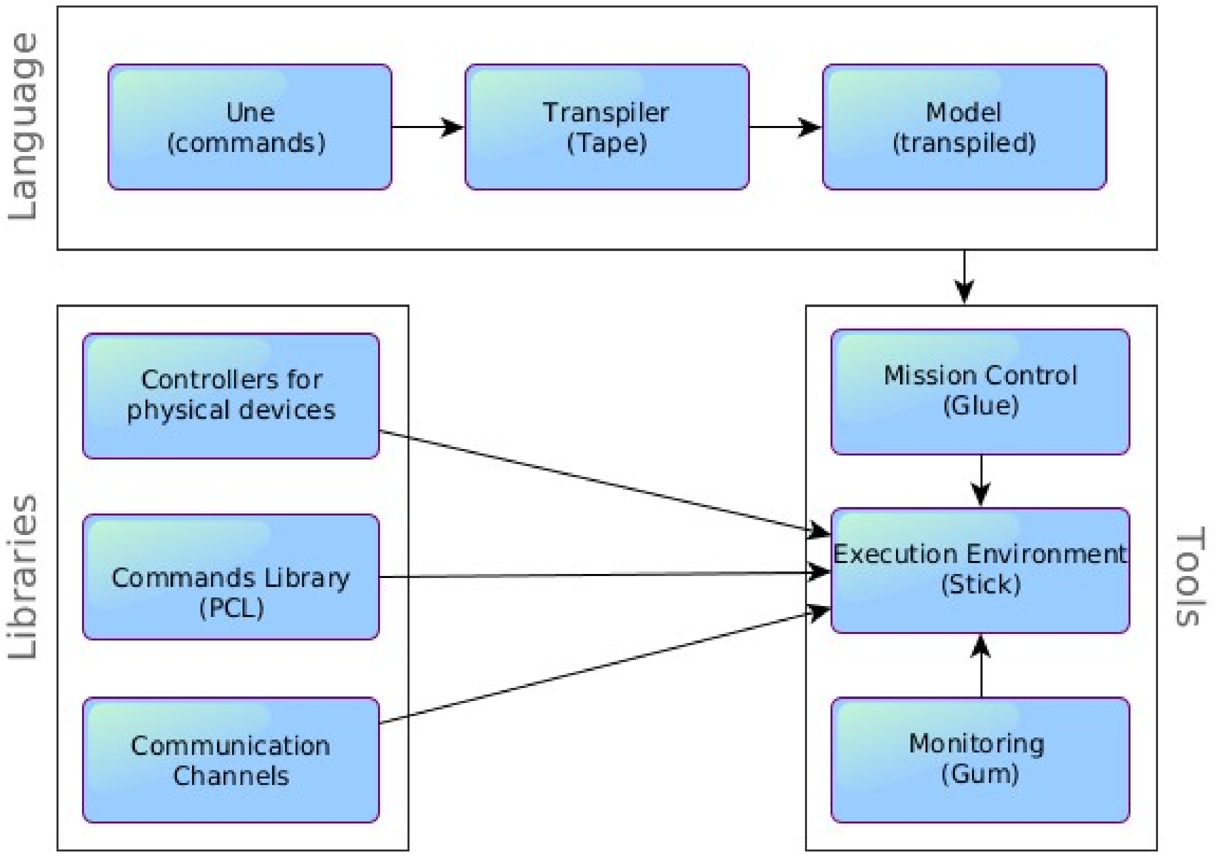
- Transpiler for the *Une* language
- Execution Environment

2 https://opensource.org/licenses/MIT
3 https://www.raspberrypi.org/

The MSP has more tools than the ones just mentioned. All of them will be discussed in this handbook. Although every tool in the Platform could have its own name, the language (among other things because it is not a tool) will be always named "*Une*", despite of the "Mingle Platform" implementation.

This table has the tools and their names in the Standard Implementation (the **MSP**):

Tool	**Name**
Transpiler	Tape
Execution Environment	Stick
Integrated Development	Glue
Monitoring / Dashboards / RESTful Bridge / File Server	Gum



Here you have a chart that represents the main components.

The “Une” language

As *Une* is the MSP cornerstone, it has its own handbook. Therefore, we will make here just a brief introduction to this language.

Une was conceived to be really concise and also as close to a natural language as possible (English, Spanish or Chinese are examples of natural languages).

Making *Une* as concise and as natural as possible, it can be learned in hours, instead of months or years that is the time normally needed to learn a traditional programming language.

One more thing before go: *Une* is case insensitive (case is ignored). In all the cases except when using strings: everything in between double-quotes will preserve its case. The locale used is the one defined at Operative System level where the Une tools are being ran.

Technical hint

If you need to change it, simply pass the desired locale when running the tool. In other others, pass it at CLI to the JVM. For instance, following will use Canadian French even if the O.S. local is another one:

```
    java -Duser.country=CA -Duser.language=fr
```

A Transpiler named “Tape”

Lets explain this tool in two different ways: one for people that do not know about programming languages and other for the people that know about it.

If you are not familiar with it

If you are not familiar with programming languages, all you need to know is that most of them are transformed (in different ways, depending on the language) before they can be used.

This tool takes a file that contains the *Une* instructions and produces another file that can be used by an Execution Environment (we will talk about this tool in next section).

The files that Tape receives have “`.une`” as extension and files produced by Tape have “`.model`” as extension.

If you are familiar with it

If you are familiar with programming languages, you need to know that Tape is not a compiler, but a “transpiler”[4].

Most compilers achieve their goal by executing following 8 stages:

1. Lexical Analysis (Scanning)
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Optimization
6. Code Generation
7. Code Optimization (Lower Level)
8. Code Emission

Transpilers perform the first 5 stages; because most of the transpilers, instead of generating code that can be executed directly by the computer (“machine code”), they generate code for another programming language (they translate one language into another). There are also transpilers that generate code that does not belong to any language, but code that will be executed by some kind of executor: an execution environment, a virtual machine, etc.

But the variety of compilers, transpilers and interpreters is so huge, the goals they pursue are so varied and the way they achieve their goals are so diverse that the lines between compilers, transpilers and interpreters are really blurred nowadays.

The code generated by the “*Mingle Platform* Transpiler” is a JSON[5] representation of the *Une* commands (the *Une* source code).

How to execute Tape

As just said, Tape takes *Une* files (normally they have “`.une`” as their extension) and produces files having “`.model`” as extension.

4 https://en.wikipedia.org/wiki/Source-to-source_compiler
https://devopedia.org/transpiler
5 https://www.json.org/

There are two ways to execute Tape:

- Using the JVM: `java -javaagent:lib/lang.jar -jar tape.jar`
- Using the script: `run.sh` or `run.bat` (explained later)

In both cases, if you do not pass a parameter or pass -help, Tape will show something similar to the following:

```
_______        ____    ____
   |      /\     |    |   |
   |     /  \    |____|   |___
   |    /____\   |        |
   |   /      \  |        |____
Version: 2024-04-22

Syntax:
      tape [options] <fileName1>[.une] ... <fileNameN>[.une]

Options (all are optional)
   -grid=true|false         Transpiles to be executed in a grid
environment.By default false.
   -charset=<a_charset>     Charset name to read local source files. By
default the platform one.
   -intermediate=true|false Prints into files the intermediate code in
'tmp' folder. By default false.
   -config=URI              URI with the configuration file to use. By
default, '{*home*}config.json'.
      -help | -h             Shows this help.

Une files can be in the following forms:
      * File name:  + With or without path and with or without extension.
                    + Wildcards and macros (Glob Syntax) can be used.
      * URI:
           + file://<file>
           + http://<uri>
           + https://<uri>


For detailed information, refer to the handbook or visit:
https://mingle.peyrona.com/docs
```

Examples of parameters that can be passed to Tape:

- `my_script_1.une my_script_2 my_script_3.une`
- `my_script_*`
- `{*home*}/examples/*.une`
- `[mM]y_script_[1-3].une`

But you are not restricted to local files: you can use also remote files, as far as the INCLUDEs commands can be satisfied; either locally or remotely.

E.g.: `tape my_script_1.une my_script_2 my_script_3.une`

Ahead in this book, you will find a more detailed description of how to pass files to Tape and other tools; under the section named “Macros and Glob Syntax”.

An Execution Environment named “Stick”

The application named **Stick** is responsible for executing the commands written in the *Une* language after they have been transpiled by the transpiler, **Tape**. As previously discussed, commands are the fundamental elements of the *Une* language, utilized to define the configuration and operational rules of a physical domotics installation. The comprehensive set of commands required to manage a single domotics installation constitutes what we refer to as a "model". This model must be loaded and executed by an Execution Environment (ExEn), which, in our implementation, is named **Stick**.

Stick warrants a detailed exposition, which is provided at the conclusion of this book. In this section, we will briefly summarize its primary objectives:

- **Execution of Transpiled Commands**: Stick executes commands that have been transpiled from the *Une* language by Tape.

- **Communication Channels for Interaction**: Stick exposes communication channels that allow listeners to interact with the execution environment. These interactions include:
    - **State Notifications**: Informing listeners when a device changes its state.
    - **Command Management**: Enabling listeners to manage the commands being executed, including the ability to:
        - Add new commands.
        - Remove existing commands.

    - Replace existing commands.
- **State Management**: Allowing listeners to query and modify the state of Stick, including:
    - Listing the current set of commands.
    - Gracefully terminating the execution.
- **Command Updates**: Notifying listeners about changes in devices.

This architecture ensures that the execution environment is dynamic and responsive, facilitating seamless management and interaction with the domotics installation.

How to execute Stick

Unlike Tape, that can process multiple files, **Stick** expects one and only one file; this file can be local or remote (via http or https).

You have two ways to execute Stick:

- Using the JVM: `java -javaagent:lib/lang.jar -jar stick.jar`
- Using the script: `run.sh` or `run.bat` (explained later)

If you pass “-help”, Stick will show the following:

```
 ____  _______  **   ____
|         |     ||  |      |  /
|____     |     ||  |      |_/
     |    |     ||  |      | \
 ____|    |     ||  |____  |  \

Stick : Mingle Execution Environment

Syntax:
    stick [[options] <fileName>[.model]]

Options:
    -config=URI               URI to the configuration file to use:
either a local file or an URI designating a remote file. By default:
{*home*}config.json.
    -log_level                Minimum log level to accept a log message.
By default "WARNING".
    -use_disk=true|false      Can create files (e.g. logs)?. y default
true.
    -clearDrivers=true|false  Remove unused drivers (CARE: those used
inside a SCRIPT will be removed: refer to handbook). By default false.
    -help|-h                  Shows this help.

Transpiled (*.model) as well as config file can be in the following
```

```
forms:
    * File name (in absolute or relative path and with file extension)
    * URI:
        file://<file>
        http://<uri>
        https://<uri>


By default, configuration file is retrieved from {*home*}. If not found,
default values are used.
Nevertheless some configuration values stored in files can be overridden
using the options above and/or passing values to the JVM.
For detailed information, visit:
https://mingle.peyrona.com/tools/stick/index.html
```

If you pass no argument, Stick will show something similar to the following:

```
Stick: An Execution Environment (ExEn) for the Mingle Standard
Implementation (MSP).

Version = 2024-04-29
Home    = /home/mingle/todeploy
Config  = Default (file:/home/mingle/todeploy/config.json)
Log     = /home/mingle/todeploy/log/stick.log.txt, Level=WARNING
Model   = Not specified
XprEval = NAJER: MSP Expressions Evaluator v.1.4
CIL     = MSP Commands Incarnation Library v.1.2
Faked   = true -> using faked drivers
Java    = version 1.11.0_261 (Oracle Corporation)
JVM     = Java HotSpot(TM) 64-Bit Server VM v.25.261-b12
JRE     = Java(TM) SE Runtime Environment. Version: 1.8.0_261-b12
OS      = Linux. Version: 6.5.0-35-generic. Architecture: amd64
Docker  = false
Locale  = en_US

Communication channels:
    * WebSocketServer: [ws://0.0.0.0:55888] [allow=intranet]
    * PlainSocketServer: [0.0.0.0:55886] [allow=intranet] [SSL=false]

Grid information:
    This ExEn does not belong to a Grid.

[2024-05-31 12:50:33] <<<<<<< Stick started >>>>>>>
```

A Mission Control named “Glue”

Glue is a GUI[6] Java application. Therefore you need a computer were Java can run in order to run Glue, but even a Raspberry Pi (PCs costing around US35$) can run Glue smoothly.

You have following ways to execute Stick:

- Using the JVM: `java -javaagent:lib/lang.jar -jar glue.jar`
- Using the script: `run.sh` or `run.bat` (explained later)
- Double-click the icon associated with “glue.jar” in your file explorer (this works only if you have Java installed in your computer).

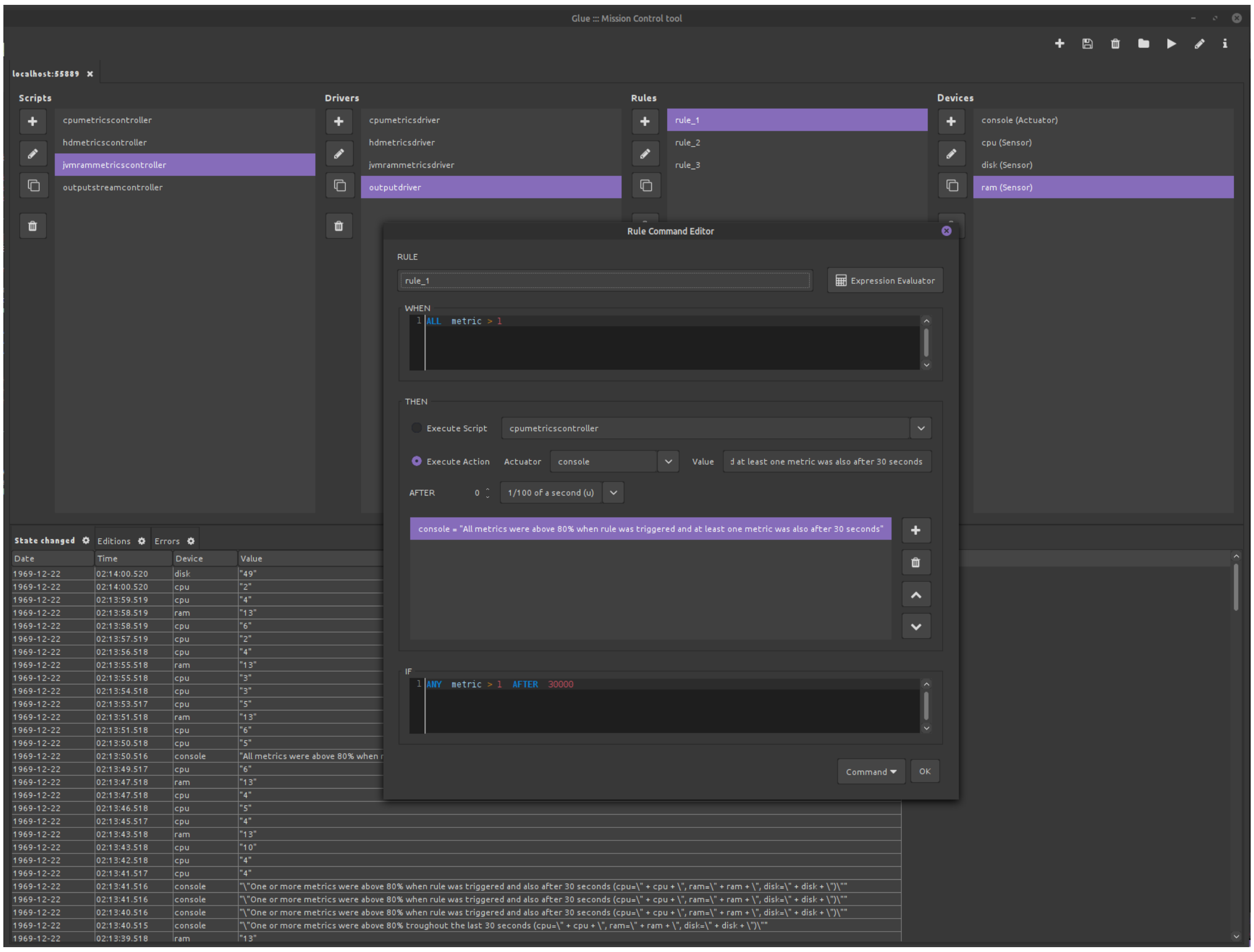
What is Glue used for?

- To connect with one or more ExEns: from inside Glue you can connect to as many ExEns as you may like.
- To monitorize an ExEn. You can
  - Inspect the set of commands that the ExEn is running
  - To receive the changes that are taking place in the devices the ExEn is running.
  - To receive the errors messages (if any) that happen inside the ExEn.
- To manage an ExEn. You can:
  - Add new commands to an ExEn on the fly (do not need to stop the ExEn)
  - Remove existing commands from and ExEn on the fly
  - Change the state of a particular device on the fly
  - Request to gently finish ExEn execution (exit).

Glue was conceived and implemented to be intuitive and user-friendly. Consequently, the most effective way to understand its functionality is through practical experimentation and hands-on use.

6 https://en.wikipedia.org/wiki/Graphical_user_interface

This is how it looks like (the look and feel may vary depending on your desktop configuration):



RULE: rule_1

WHEN

```
ALL  metric > 1
```

THEN

Execute Script: cpumetricscontroller

[✓ Execute Action] Actuator: console Value: d at least one metric was also after 30 seconds

AFTER 0 1/100 of a second (u)

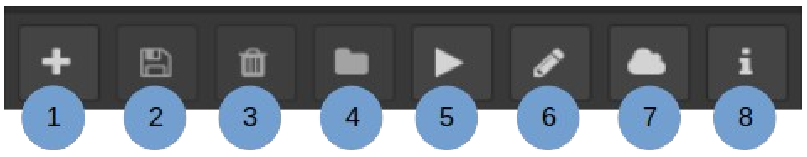
console = "All metrics were above 80% when rule was triggered and at least one metric was also after 30 seconds"

IF

```
ANY  metric > 1  AFTER  30000
```

Date	Time	Device	Value
1969-12-22	02:14:00.520	disk	"49"
1969-12-22	02:14:00.520	cpu	"2"
1969-12-22	02:13:59.519	cpu	"4"
1969-12-22	02:13:58.519	ram	"13"
1969-12-22	02:13:58.519	cpu	"6"
1969-12-22	02:13:57.519	cpu	"2"
1969-12-22	02:13:56.518	cpu	"4"
1969-12-22	02:13:55.518	ram	"13"
1969-12-22	02:13:55.518	cpu	"3"
1969-12-22	02:13:54.518	cpu	"3"
1969-12-22	02:13:53.517	cpu	"5"
1969-12-22	02:13:51.518	ram	"13"
1969-12-22	02:13:51.518	cpu	"6"
1969-12-22	02:13:50.518	cpu	"5"
1969-12-22	02:13:50.516	console	"All metrics were above 80% when
1969-12-22	02:13:49.517	cpu	"6"
1969-12-22	02:13:47.518	ram	"13"
1969-12-22	02:13:47.518	cpu	"4"
1969-12-22	02:13:46.518	cpu	"5"
1969-12-22	02:13:45.517	cpu	"4"
1969-12-22	02:13:43.518	ram	"13"
1969-12-22	02:13:43.518	cpu	"10"
1969-12-22	02:13:42.518	cpu	"4"
1969-12-22	02:13:41.517	cpu	"4"
1969-12-22	02:13:41.516	console	"\"One or more metrics were above 80% when rule was triggered and also after 30 seconds (cpu=\" + cpu + \", ram=\" + ram + \", disk=\" + disk + \")\""
1969-12-22	02:13:41.516	console	"\"One or more metrics were above 80% when rule was triggered and also after 30 seconds (cpu=\" + cpu + \", ram=\" + ram + \", disk=\" + disk + \")\""
1969-12-22	02:13:40.516	console	"\"One or more metrics were above 80% when rule was triggered and also after 30 seconds (cpu=\" + cpu + \", ram=\" + ram + \", disk=\" + disk + \")\""
1969-12-22	02:13:40.515	console	"\"One or more metrics were above 80% troughout the last 30 seconds (cpu=\" + cpu + \", ram=\" + ram + \", disk=\" + disk + \")\""
1969-12-22	02:13:39.518	ram	"13"

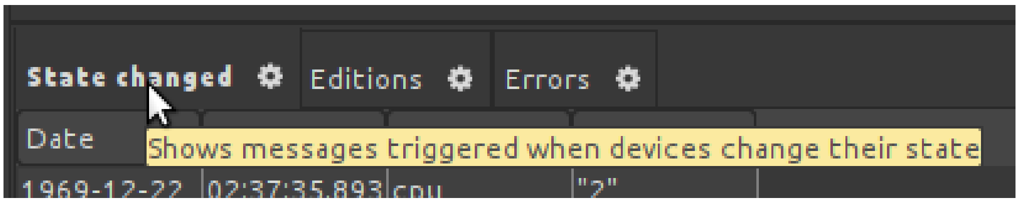
Lets now take a look to the tool bar buttons (the look and feel may vary depending on your desktop configuration):



1. Opens a dialog window to connect to an existing ExEn
2. Saves the set of commands that an ExEn is running into a "`.model`" file
3. Empties the associated ExEn (removes all commands)

4. Loads a "`.model`" file and send all commands to an ExEn
5. Starts an ExEn in the same machine were Glue is running
6. Opens a text editor designed to work with "`.une`" files
7. Starts/stops **Gum**
8. Shows a dialog window with information about Glue

When the mouse pointer is not moved for around a second over a component, Glue shows a tooltip. This is an example:



A Swiss-knife web-tool named "Gum"

Gum is used to accomplish following tasks:

- To visualize in real-time the state for devices running inside an ExEn[7] using RESTful services.
- To change in real-time the state of devices running inside an ExEn using RESTful services.
- To visually (WYSIWYG[8]) manage (CRUD[9]) dashboards that show the state and optionally change the state of devices running inside an ExEn.
- To visually (using a HTML5/CSS3/JavaScript6) manage a remote (server-side) folder-tree to server static contents (files).

Gum is a web tool with two sides:

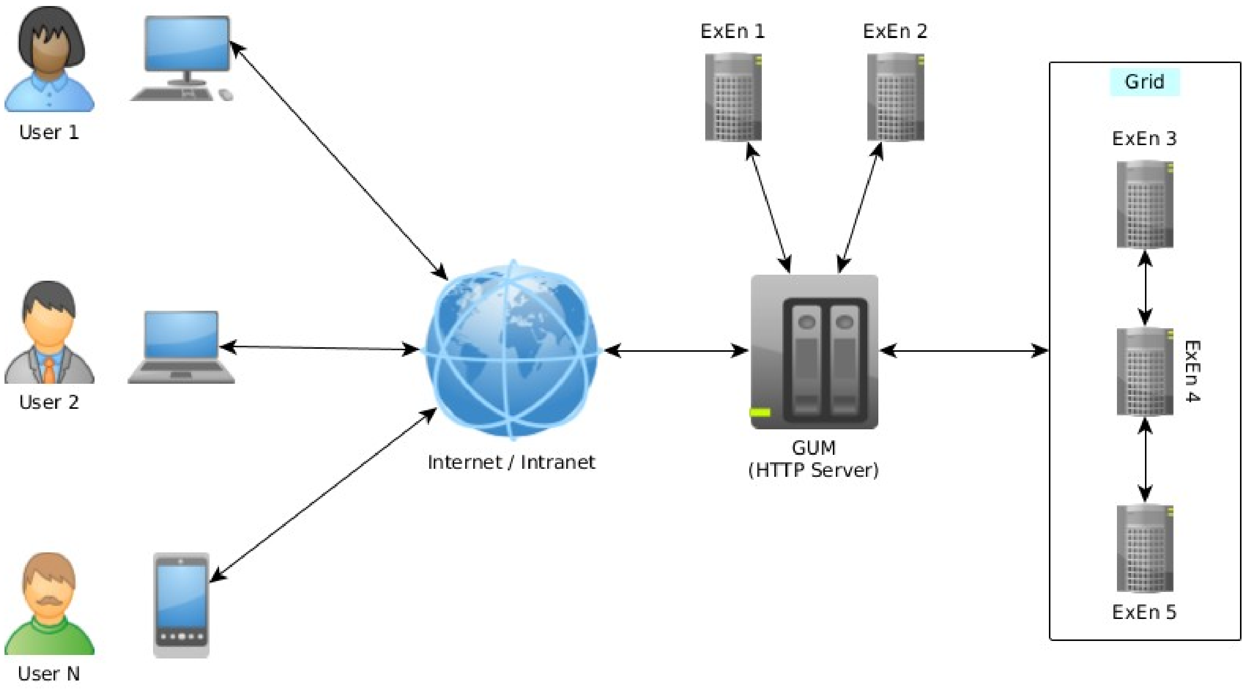
- The server side, written used Java EE 7 and running using embedded Undertow (JBoss/Redhat/IBM) server. Although any "Servlet Container" that conforms with (v.g. Tomcat 8[10] and above) can be used to run this source Java EE code.

7 *Stick* if you are using the Mingle Standard Platform
8 What You See Is What You Get
9 Create, Read, Update, Delete
10 https://tomcat.apache.org/tomcat-8.0-doc/html-manager-howto.html

- The client side: any web browser having JavaScript ECMA 6 or above can be used. Any modern web browser (FireFox, Chrome, Safari, etc) should work. For security reasons, the client side does not connect with the ExEns, but with the RESTful services, which "talks" with the ExEns. So you do not need to expose the ExEns to Internet and you can use all the security facilities that a modern Web-Server provide.

Gum can run in any modern web browser in a PC, tablet or smart phone. As long as you have a web browser and an Internet connection, **Gum** can be used to remotely access, monitorize and operate with ExEn instances, from any place and in a safe way. **Gum** can be configured using "config.json" file. The module named "monitoring" provides specific **Gum** parameters.

Following diagram shows -in a simplified way- how it works. Meanwhile communications at the left side of the "HTTP Server" icon are done using "Web Socket" protocol, communications at the right side can be done using any other available protocol declared in MSP (by default "Plain Socket"); therefore it is very important that all ExEns that are accessed by the "HTTP Server" implement this way of communicating.



When it is started an output similar to next one:

```
[2024-06-07 09:48:54] starting server: Undertow - 2.3.10.Final
[2024-06-07 09:48:54] XNIO version 3.8.12.Final
[2024-06-07 09:48:54] XNIO NIO Implementation Version 3.8.12.Final
```

```
[2024-06-07 09:48:54] JBoss Threads version 3.5.1.Final
[2024-06-07 09:48:54] Gum started. Context: http://localhost:8080/gum/
[2024-06-07 09:48:54] Web folder is: /home/mingle/todeploy/lib/gum/web
[2024-06-07 09:48:54] Serving files from:
/home/mingle/todeploy/etc/gum_web_user_files
[2024-06-07 09:48:54]     * at context: http://localhost:8080/gum/user-files
[2024-06-07 09:48:54]     * UI manager:
http://localhost:8080/gum/file_mgr/file_manager.html
[2024-06-07 09:48:54] Is inside docker: false
```

When talking about dashboards, **Gum** has two modes: design and execution. In the first mode it allows to manage visualizations (named Dashboards) and also to "run" the visualizations. The second mode is a "read-only" mode (from the point of the view of the dashboards): you can interact with them but you are not allowed to modify them.

Some tasks that can be performed using **Glue** can also be achieved using **Gum**, but meanwhile **Glue** is development oriented, **Gum** is visualization oriented. In other words, meanwhile **Glue** is more oriented to manage (to add, to remove and to modify) commands running inside an ExEn, **Gum** is more oriented to visualize in real time the state (the value) of the devices that are running inside an ExEn.

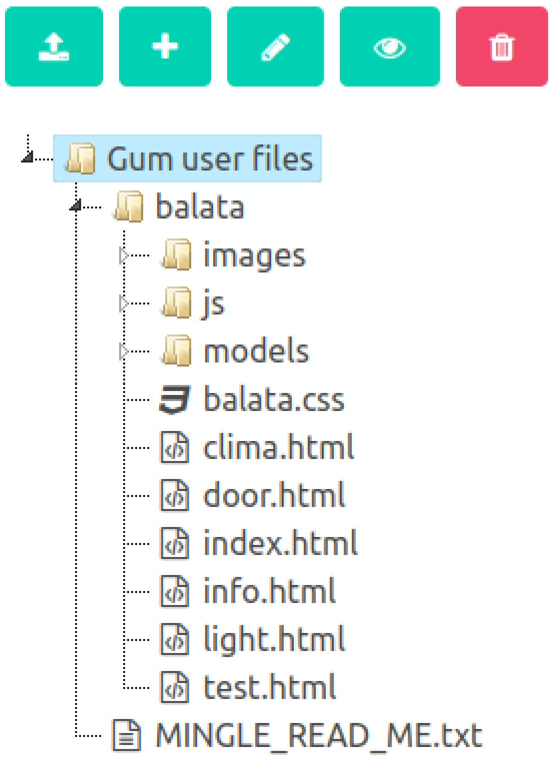
Gum also allows to change the state of a device by sending a message to the ExEn. In this way (from a web page) you can for instance switch on and off a light or an air conditioned. **Gum** will send the appropriate instructions to ExEn and the device will change its state.

HTTP Server for user files

HTTP file server that enables users to create and manage directories, as well as upload and delete files within these directories. The server is specifically designed to handle static content, such as HTML, CSS, and JavaScript files, and does not support dynamic content execution (e.g., Java, PHP). Additionally, a JavaScript-based graphical user interface (GUI) has been developed to facilitate these operations, providing an intuitive and user-friendly experience for users.

The primary objective is to provide a secure and efficient platform for managing static files, ensuring ease of use through a dedicated JavaScript GUI.

This is how the GUI looks like.



Architecture

The system architecture of the HTTP file server is divided into two main components: the server-side application and the client-side graphical user interface.

1. **Server-Side Application**
   - **Technologies Used**: Java EE 7
   - **Functionality**: The server-side application handles HTTP requests to create, delete, and manage directories and files. It enforces strict file type restrictions to ensure that only static content (HTML, CSS, JavaScript) is allowed.
   - **Security Measures**: The server implements security measures to prevent the execution of dynamic content and mitigate potential security vulnerabilities.
   - **Endpoints**:
     - **GET**: Retrieve the list of files and directories, or the contents of a specific file.
     - **PUT**: Create new directories.
     - **POST**: Rename files or directories.
     - **DELETE**: Delete specific files or directories.
2. **Client-Side Graphical User Interface**
   - **Technologies Used**: JavaScript, HTML, CSS
   - **Functionality**: The GUI provides an interactive interface for users to perform file management operations. It communicates with the server-side application using AJAX requests.

- **User Experience**: The GUI is designed to be intuitive, allowing users to easily navigate the file system, upload files, and manage directories without requiring technical expertise.

Implementation Details

The implementation details of the HTTP file server include the following key aspects:

1. **Directory and File Management**
    - The server supports the creation and deletion of directories. Directory paths are validated to prevent unauthorized access to the file system.
    - Files can be uploaded to the server, with restrictions in place to ensure that only HTML, CSS, and JavaScript files are allowed. This is enforced through MIME type checking and file extension validation.
2. **Security Considerations**
    - The server includes mechanisms to prevent the upload of executable files or scripts that could pose security risks.
    - User authentication and authorization are implemented to control access to the file management features.
3. **GUI Functionality**
    - The JavaScript GUI interacts with the server using XMLHttpRequest or Fetch API to perform file management operations asynchronously.
    - The interface includes features such as drag-and-drop file upload, directory tree navigation, and context menus for file operations.

RESTful Bridge to ExEns

Dashboards

This is how a dashboard looks like while in design mode (the look and feel depends on the web-browser you are using):

Poner aqui una captura de pantalla

You design a dashboard by adding visual components (a device visualizer). or can be Following visualizers are currently available:

- **Charts** can show the evolution of one or more Sensors values in time. Sensors has to be of type numeric, for instance thermometers in different rooms. It can also show historic data from a DataBase using any proper JDBC connection.
- **Displays** are the old fashion “seven segments displays”; and are used to show the current value of their associated Sensor. Obviously, the sensor has to be of type numeric (v.g. a thermometer).
- **Check-boxes** are used to show bi-state (boolean) Sensors (v.g. a motion sensor) state and Actuators (v.g. a on/off lamp) state. They can be used also to change the state of an Actuator. You can associate a custom image to the ON state and another image to the OFF state.
- **Push-buttons** are used to change the state of one or more Actuators. All these actions will be executed every time an user clicks the button.
- **Texts** are used to show Actuator’s value of type text (strings). But if you do not “connect” the visalizer to an Actuator, the Texts can have a fixed text: in this case, Texts act as Labels (or Titles, as you prefer to name it).
- **Images** are used to show a specific image. Note: background images can be set separately.

After you deployed **Gum** in a “HTTP Server”, let say at `192.168.1.222`, you can access Gum by writing this at your web browser: `http://192.168.1.222/gum` or `https://192.168.1.222/gum`, depending how you configured your “HTTP Server”.

Gum is really intuitive, so the best way to learn how to work with Glue is playing with it. You have a dashboard to play with it named `Basic_Example.html` (this is the screen shown above). To feed the dashboard with data, you have to run an ExEn with “`99.gum-basic-example-values-generator`”.

To run this example, follow theses steps:

1. Run ExEn: `stick {*home*}examples/90.gum-basic-example.model`
2. Run the Servlet Container (v.g. Tomcat) where Gum (gum.war[11]) is deployed.
3. Open a modern WebBrowser and point to the URL where Gum indicates.
4. Select “Basic_Example” dashboard
5. Learn how was it built and make changes.

11 You can find it under: `{*home*}lib` folder

RESTful services for Dashboards

Cogerlo de mi tesis

MSP project workflow

In this section will show how to create and fine tune a project in IS. This approach perhaps is not the one that better fits your way of doing things. There is not only one way of doing it and this proposal could be not the best workflow, but it is the one we use.

Laksdjflkasjdflkjasdlfkj

How ExEn works

Messaging

Asynchronouscity

This is a consequence of what is explained in Messaging: because messages can be processed in parallel, those accomplishing their task in less time will send new messages to the bus (triggering new actions) earlier than others.

This implies that we can not guarantee the execution order of a rule's actions. The order they appear in an *Une* script, could or could not be the order they are executed.

A question of time

Managing time consist basically in:

a) Triggering events at a precise moment: date and time.

b) Measuring intervals: time elapsed between two moments.

Initial note: different implementations can manage different time precisions: in other words, meanwhile certain implementation could not be able to manage magnitudes of time shorter than milliseconds (1/10^3 seconds), other implementations could manage microseconds (1/10^6 seconds) or even nanoseconds (1/10^9 seconds). It is sat that all implementations should manage at least an accuracy of millisecond.

The Exceutable Environment (ExEn) works asynchronously

This how Mingle handles time (as an advance: 'Language' class is in charge of it):

Language::set

First Example

Lets say we want to switch on out garden lights everyday when it is getting dark. This time is not the same through all the year, but fortunately, *Une* provides a way to know it, so we can create a **`RULE`** similar to this one:

```
WHEN clock_dawn AND get("garden_flag") IS OFF \
                AND time() > time():twilightSunSet()
THEN garden_lights SET ON
     put("garden_flag", ON )
```

There is an additional point we have to be highlighted here: because `get("garden_flag")` is an operation that consumes much less CPU[12] than `time() > time():twilightSunSet()`, we place it at first; so, when the first **`AND`** is not `TRUE`, the second **`AND`** will not be evaluated.

A complete script containing all needed code can be found at: `{*home*}examples/garden_stuff.une`

Drivers

Drivers (keyword `DRIVER`) are the link between the logical devices (keyword `DEVICE`) and the physical devices.

12 A simple rule of thumb: `date`, `time`, `list` and `pair` related operations are more expensive (in terms of CPU consumption) than the rest.

Drivers are similar to the Operative System (Windows, Linux, etc) drivers: your OS needs a driver to manage the physical devices: printer, scanner... In the same way, the *Mingle Platform* needs Drivers to manage physical devices (Sensors and Actuators).

In order to print in your printer you do not need to know how the driver works: the manufacturer that fabricated your printer also fabricated its driver. This is the same with *Mingle* Drivers[13]: you do not need to know how they work in order to manage them. You just need to know the parameters a Driver manages and the type of data these parameters accept.

A Driver for a Sensor, reads a value from a physical gadget (e.g. a thermometer) and updates the value of the Sensor with it. Every time the thermometer changes its value, the associated Sensor's value is updated and all Rules are re-evaluated.

A Driver for an Actuator reads the value from a physical gadget, but I ca also change the estate of the physical gadget. Every time the Actuator changes its value, the associated Driver will change the state of the physical gadget, the light is turned ON or OFF. And all Rules are re-evaluated.

All drivers delivered with the MSP are safe: they do not allow to execute O.S. commands and when they deal with files, you can always tell the name and the place where the files will be stored; in this way you can always monitorize them.

The *Mingle Standard Platform* (MSP) is delivered with a bunch of drivers ready to be used. All of them are described in this file: `{*home*}include/common-drivers.une`. To inform to *Une* that you want to use these drivers, you have to use the `INCLUDE` command to effectively include this file along with the rest of your commands. It is strongly recommended to take a look to the file to know what parameters to use with the `DRIVER` command.

For more information about every driver, please read files:

`{*home*}include/`

There are examples covering MSP drivers at following folder:

{*home*}examples/

Following are the files that contain the drivers provided by MSP at the moment of writing this document.

- basic-drivers.une
- daikin-drivers.une

13 Drivers can be created by volunteer people instead of hardware manufacturers. At the end, for you there is no difference.

- lednet-drivers.une
- microsoft-drivers.une
- protocol-drivers.une
- voice-drivers.une
- web-services-drivers.une

Each file is meticulously documented, and at least one example is provided in the examples folder, demonstrating the fundamental operations of the drivers.

How to create a Driver

The objective of this document is not to provide a comprehensive description of the process of developing a driver from the ground up. However, developers proficient in the Java language may consult the drivers provided by the MSP.

It is important to note that drivers can be developed using any language that Mingle will accept. In the case of MSP, these are: Java, Python, and JavaScript.

Other modules

IMPORTANT:

> This section is for advanced users and it is not needed to create and manage fully operational domotics installations using the MSP: if you are not an experienced programmer, you can simply ignore this section.

What we have seen until now are the main modules that comprise the *Mingle Standard Platform* (MSP), but there are more. The complete list of modules is out of the scope of this handbook, but lets take an overlook to few others.

All modules described in this section are plugable. It means they are loaded at run time. This allows to replace the module delivered with the MSP by a different module created by third parties; others having perhaps more features or just simply a module that you feel more comfortable with.

To specify which module is going to be loaded at run time, we have to edit the "`config.json`" file (covered later). Once the module is loaded, the rest of modules that comprises the MSP will remain the same: the change will be transparent for the rest of the MSP.

NetWork

As we said when talking about the ExEns, other tools (like Glue and Gum) can connect with them using several protocols. **Network** is the module that provides the protocols for these communications.

The MSP is delivered with the following protocols:

- Plain Socket: used mainly in Intranets
- Web Socket: used mainly when connection via Internet

Compilers and Interpreters (candi)

Programmers can write `SCRIPT` actions using the languages that the *Mingle Platform* provides. As in many other cases, different *Mingle Platform* implementations would offer different languages. The MSP offers: Java, Python and JavaScript (more will be added in the future).

The **candi** module takes care of compiling, transpiling and interpreting the set of languages available in the MSP, including the *Une* language.

Controllers

As it was explained before, drivers are the link between virtual devices and the physical devices. But this is not completely true: in this link there is a piece in the middle, yes the controller. To be precise, a Driver is a link between a Device (Sensor or Actuator) and its Controller, and a Controller is a link between the Driver and its physical device.

Controllers are the last responsible entity that manage physical devices. Anyone can create a Controller and its Driver, but ideally they are created by the hardware manufactures that fabricate the physical devices; therefore, how to create them is a topic out of the scope of this handbook.

Controllers are similar to the Operative System drivers: your OS needs a driver to manage the physical devices: printer, keyboard, scanner... In the same way, the *Mingle Platform* needs Controllers to manage physical devices (Sensors and Actuators).

In order to print in your printer you do not need to know how the driver works: the manufacturer that fabricated your printer also fabricated its driver. This is the same with controllers: you do not need to know how they work in order to manage an air conditioned or a dimmed light.

As an example: lets say ACME company creates a new “dimmerizable bulb light” that can be managed via WiFi. Lets say they name this product “Acme Bulb”. People can change its intensity (its "candela[14]"). ACME would create a controller to be used by the Mingle Platform (any implementation). This controller will act as the printer driver that you need to use your printer. ACME then has one more thing to do: they have to declare how to use their controller (this is what the *Une* command `DRIVER` is used for). This last task is trivial and it will take to ACME no more than 3 to 7 minutes.

The driver will depend on how the controller works, but lets say the controller has only one parameter to control the candela in a range from 0 to 100. So, the driver would look like this:

```
DRIVER AcmeBulbDriver
    SCRIPT AcmeBulbController
    CONFIG
        candela AS number    # From 0 to 100 (dimmer)
```

And you could declare a bulb like this:

```
ACTUATOR bedroom_light
    DRIVER AcmeBulbDriver
```

NAJER

This is the default MSP Expressions Evaluator (in the "`config.json`" file, an Expressions Evaluator is referred as "ExprEval").

An Expressions Evaluator is always a key module in any language, but it is specially important in *Une* because in *Une* the system logic resides in `RULE`s and rules only do 2 things: to evaluate expressions and to execute actions.

In the *Une* handbook you will find a whole chapter dedicated to NAJER.

By the way, NAJER is the acronym for: Not Another Java Expressions Resolver.

Commands Library

In the MSP, this module is known as CIL (Command Incarnation Library).

The MSP is so flexible that even the *Une* commands module can be substituted by another. It is true that this is a delicate task, but it is possible.

Replacing this module is important, for instance, when someone wants to have the Mingle Platform working for an OS were Java does not work: Tape, Glue and Gum will remain the same, but Command Incarnation Library (among other things) has to be rewritten.

14 https://en.wikipedia.org/wiki/Candela

Macros and Glob Syntax

Both allow to work with files in MSP.

Macros

Following macros (once again: case is ignored) can be used when we want to refer to files that are stored in the MSP predefined folders: they are frequently used in conjunction with some of the files explained later.

Macro	Meaning
`{*home*}`	Where the MSP is installed
`{*home.lib*}`	Where the additional MSP libraries are
`{*home.log*}`	Where log files are
`{*home.tmp*}`	Used for temporary files: its contents can be safely deleted
`{*home.etc*}`	Where any other files go (a miscellaneous folder)
`{*nl*}`	New line (NL, RT or NL & RT, depending on the OS)

These macros can be used whenever a file is going to be used (all occurrences of the macro will be replaced). Example:

```
INCLUDE "file://{*home*}include/standard-includes.une"
```

Note: do not include a back-slash (‘/’) after a macro:

- WRONG: `"file://{*home*}/include/standard-includes.une"`
- RIGHT: `"file://{*home*}include/standard-includes.une"`

Glob Syntax

You can use a simplified glob syntax to specify pattern-matching behavior (which files a tool will process).

If you have ever used a shell script, you have most likely used pattern matching to locate files. In fact, you have probably used it extensively. If you haven't used it, pattern matching uses special characters to create a pattern and then file names can be compared against that pattern. For example, in most shell scripts, the asterisk, *, matches any number of characters.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names. Glob syntax follows several simple rules:

- An asterisk, *, matches any number of characters (including none).
- Two asterisks, **, works like * but crosses directory boundaries. This syntax is generally used for matching complete paths.
- A question mark, ?, matches exactly one character.
- Square brackets convey a set of single characters or, when the hyphen character (-) is used, a range of characters. For example:
- [aeiou] matches any lowercase vowel.
- [0-9] matches any digit.
- [A-Z] matches any uppercase letter.
- [a-z,A-Z] matches any uppercase or lowercase letter.
- All other characters match themselves.
- To match *, ?, or the other special characters, you can escape them by using the backslash character, \. For example: \\ matches a single backslash, and \? matches the question mark.

Here are some examples of glob syntax:

- *.une – Matches all strings that end in .une
- ??? – Matches all strings with exactly three letters or digits
- *[0-9]* – Matches all strings containing a numeric value
- a?*.une – Matches any string beginning with a, followed by at least one letter or digit, and ending with .une

You can learn more about the Glob Syntax in many places, but this explanation was based in another one found here:

https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob

Other important files

This section is for advanced users: unless you know what you are doing, you better do not mess with the files mentioned in this section.

run.sh and run.bat

This very simple script main purpose is to show how to start the MSP tools.

run.sh is meant to be used in Linuxes and Macs and **run.bat** is meant to be used in Windows.

Paths in these files are relative to location of the file (Mingle home AKA base dir), therefore in order to work properly, these files must be executed from where they are located.

Obviously, you can use it to run the tools; but besides this, we suggest for you to open this file (.sh or .bat depending of which are you more familiar with) with a text editor and learn about it[15].

config.json

This JSON encoded file is divided into following modules:

- Common
- Transpiler
- ExEn
- ExprEval
- CIL
- CandI
- Network

Most part of the entries found in the configuration file were already somehow explained, lets now explain the entries we did not cover yet:

Module Common

`log_level`: defines a set of standard logging levels that can be used to control logging output. This value can be set either using the configuration file or via CLI (Command Line Interface); passing following option when starting Stick:

```
-Dlog_level=<a_level> (e.g.: -D=log_level=RULE)
```

The logging levels are ordered: enabling logging at a given level also enables logging at all higher levels. The default is "`WARNING`".

15 Files have to be marked as executable and you need a console and privileges to run them.

Messages are logged into files, which are located at `{*home.log*}` folder. Depending of Level granularity, the files can grow slower or faster.

In the case of the ExEn (tool Stick in the MSP), messages are also sent to all connected clients, so the network load will also depend on level granularity. To avoid network overload, only `SEVERE` and `WARNING` are broadcasted.

The levels in descending order are (names are case insensitive):

- `SEVERE` (highest value)
  Only logs internal errors: something went wrong and most probably some functionality is missed.

- `WARNING`
  Logs internal conflicts: something is not going fine, but most probably functionality is intact. Also logs previous (SEVERE) messages.

- `INFO`
  Logs information about starting and ending information of the tools plus some special information about the changes in their state. Also logs all previous messages.

- `RULE`
  Logs information about RULEs activity. It is used mainly to debug rules (in other words: to find out why a RULE is not behaving as we expected). Also logs all previous messages.

- `MESSAGES` (lowest value)
  Shows all messages that are circulating in the ExEn. It is used mainly to debug the *Une* code. Be careful, this can be very verbose: the log file will grow quickly. Also logs all previous messages.

There is an additional level: `OFF`, that can be used to turn off logging.

`faked_drivers`: when true the drivers that follow this directive will generate faked values instead of asking the physical device for its value. This is very convenience for debugging purposes because you do not even need to own (buy) the physical device in order to test it. By default it is false.

Passing from CLI "-Dfaked_controllers=true" or setting it to true in the configuration file, gives to the drivers a hint (drivers can use it or not) to generate arbitrary (faked) values instead of accessing the physical world to retrieve real values: this is very useful at development and debugging stages. This value can be set either using the configuration file or via CLI (Command Line Interface); passing following option when starting Stick:

```
        -Dfaked_controllers=true
```

TERMINAR LOS MÓDULOS QUE FALTAN

An IP is considered local if it is in same chunk (same 4th IPV4 group).

Stick at a glance

Lets talk a little bit more about the Execution Environment (ExEn).

When we use the term “ExEn”, we refer to any ExEn implementation and when we say “Stick” we are referring to the particular ExEn that is provided by the MSP.

Executing commands

This is main ExEn’s purpose.

Communication channels

To interact with an ExEn, you need to communicate with it; and to do so, you need to declare at least one *communications channel*. This is done using the “`config.json`“ file, which was covered previously. As we saw, in this file it is specified were is the native code that is in charge of communicating ExEn with the world; ExEn will load this native code to be able to receive requests and send information. This “`config.json`“ file also has the configuration parameters that ExEn has to pass to the native code after been loaded.

In the particular case of Stick most probably you will use the TCP/IP protocol (this is the default one), but you could use any communications protocol as far as you have a Java implementation of it; for instance, you could use Bluetooth[16] or Zigbee[17] or any other.

If you decide to use TCP/IP protocol to connect with Stick, there is no extra work to do because “`config.json`“ file is already prepared to do so and the protocol code is provided along with the MSP.

In fact, MSP offers 2 ways to communicate with Stick (both run over TCP/IP):

- **Sockets**: these are the old traditional sockets.
- **WebSockets**: useful to communicate with Stick from a WebPage.

16 https://en.wikipedia.org/wiki/Bluetooth
17 https://zigbeealliance.org/

How to send requests

As we have seen, you can use **Glue** and **Gum** to easily communicate with **Stick**, and this is the preferred way. Now we will see how it is done internally.

To send a request to an ExEn, you send a plain text string with following JSON format:

```
{"<request>":<payload>}
```

Where request is one of following strings (it is case-sensitive): `List`, `Add`, `Remove`, `Change`, `Exit`.

The ExEn responds with one or more messages where each message is a plaint text JSON with same format as the request. The "request" will be the same as the one received (`List`, `Add`, etc.) and the "payload" will depend on what was requested.

Following table shows the options:

Request	Payload

List	Payload to send:
	`{“List”:null}`
	
	What is `returned`:
	A JSON array with all commands in use: one command per array item
	
	MSP Java:
	`new ExEnComm( Command.List, null )`

Add	Payload to send: One or more (in a JSON array) transpiled commands to add to current set of commands. What is returned: One message per command added.
Remove	Payload to send: A JSON array with one or more command's name(s) to remove from current set of commands. What is returned: One message per command removed.
Read	Payload to send: `{"Read":"<device_name>"}`

	What is returned: `{“Read”: "<device_name>", "value":<new_value>}`
Change	Payload to send: `{"Change":"<device_name>", "value":<new_value>}` Explanation: Changes an Actuator state

Exit	Payload to send:
	`{“Exit”:null}`
	
	What is returned:
	Nothing
	
	Explanation:
	Gracefully ends ExEn execution
	
	MSP Java:
	`new ExEnComm( Command.Exit, null )`

Messages ExEn send to its clients

ExEn also sends back to all its listeners a message every time a device changes its state or an error happens.

Report	Payload
`Changed`	Payload to be received: `{"device":"<device_name>", "value":<new_value>}`

`Error`	Payload to be received:
	`{“Error”:<”description”>}`
	
	Explanation:
	`Informs that an error happened inside the ExEn.`

So, in order to receive information about the changes in the state of the devices that are running inside an ExEn, all we need to do is to connect to this ExEn. As soon as a device (Sensor or Actuator) changes its state, we will receive a text message in JSON format as described in previous table.

The same is valid also for errors.

Adding and Removing widgets at run-time

WARNING: this feature is only for advanced users.

You can add and remove Sensors, Actuators, Drivers and Rules at run-time (meanwhile the ExEn is doing its job). In fact this is what Glue does. But this is a delicate task: with great power comes great responsibility.

The details of this is out of the scope of this handbook, but you can take a look to Glue source code to find out how it is done.

Following there are just two out of the many considerations you have to keep in mind prior to add or remove components at run-time:

- When adding a new Device (sensor or Actuator), existing groups will not affected by the groups (if any) that the new Device belongs to.
- Prior to remove a Device (sensor or Actuator), all associated Rules have to be manually removed.

Grid of Sticks

Overview

Multiple instances of the Stick (ExEn) can be interconnected to operate collaboratively, forming a grid or hive of Sticks. Une scripts can be decomposed to the command level, allowing individual statements of the installation logic to reside in different ExEns. For instance, three devices can each be declared in separate ExEns, and all three devices can be managed by a single rule that resides in another ExEn. This modular approach enhances the flexibility and scalability of IoT system management.

Porque un device puede formar parte de una o varias reglas en varios ExEns, hay que informar de todos los cambios que se produzcan en los DEVICEs a todos los ExEn de un mismo Grid.

Pero es responsabilidad del programero que los nombres de todo (DEVICE, RULE, etc) sean únicos en el Grid, porque algunos mensajes, una vez que se han ejecutado para un comando, ya no se reenvían a otros nodos del Grid (ver: Stick::Orchestrator::broadCast()

Configuration

How it internally works

```
//        // Driver adds itself to listen MsgChangeActuator messages: when an Actuator changes its newValue in the Logical Twin,

//          // the Actuator posts a message of this class into the bus. Driver gets it and sends it to the Controller.

//          // If everything goes fine at Controller, the Controller will later send back a message indicating the new value of

//        // the Actuator. But if something fails at the Controller, the Controller sends an error (listener:onError(...))

//        rt.bus().add( lstnrChange, MsgChangeActuator.class );

//

//        // Driver adds itself to listen MsgReadDevice messages (e.g. when ExEn starts, devices send this request to find out

//          // their initial newValue). When a message of this class arrives, the Driver passes it to the Controller which will invoke

//             // its Listener:onChanged(...) passing to it the device's deviceName and newValue.

//        rt.bus().add( lstnrRead, MsgReadDevice.class );
```