Liquid-IoT API Specification

Tampere University of Technology, Pervasive Computing Department

25/05/2016

1. Introduction

Although, at the time being, many IoT devices have limited resources (e.g., memory, storage, processing, etc.), in the near future even the simplest things (like Raspberry Pi) leverage enough resources to host and run programs. Based on this assumption, we designed Liquid-IoT Runtime Environment (LIoTRE) to make an IoT device capable of hosting IoT applications by turning it into an application server. LIoTRE exposes its functionalities to host and manage IoT application through RESTful web services. This document provides a description of the LIoTRE API.

2. Overview on the Liquid-IoT Runtime Environment API

The main resource in L-IoTRE API is *Application*. Figure 1 shows the basic usage scenario to run an IoT application through LIoTRE API:

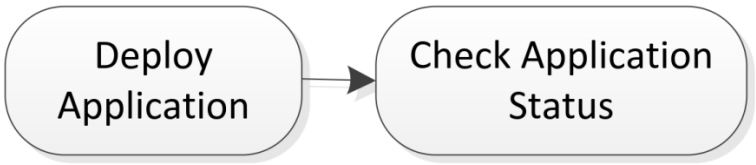

Figure 1 - Application Deployment scenario

3. The Application Resource Management Methods

In this section, we present methods to manage an application resource using REST architecture. They include: deploying application, updating application, deleting application, listing applications, deleting applications, getting application log, starting application, and stopping application.

3.1. Deploying Application

This method creates a new application. When an application is being created, its status is "initializing". Application creation request leads to a response with application description (including application "id" and "status") in JSON format. To know if the application was started successfully or not, get the application description (see Subsection 3.5) and check its "status" key. If the application can start successfully its status will be "running". If the application cannot start (e.g., due to a bug), its status will be "crashed".

The content of the request should be the application package in tarball in .tgz format. One way to pack the application in tgz format is using "npm pack" command.

The application should have certain file structure and files. The file structure of the application is described in Section 4 *File structure of the LIoT application* in details.

The response is the application description in JSON format. For more details about application description see Subsection 3.6 *Getting application description*.

Resource identifier	/app
HTTP method	POST (multipart/form-data)
Input Parameter	The tarball of the application.
Response	Application description (including Application ID and status) in JSON format.
Status code	200 if OK, otherwise the error code

3.2. Updating Application

This method updates content of an existing application. The application is identified by its ID (i.e. appId) and the new application package (tarball in .tgz format) must be provided as the input parameter.

Resource identifier	/app/{appId}
HTTP method	POST (multipart/form-data)
Input Parameter	The tarball of the application.
Response	Application description.
Status code	200 if OK, otherwise the error code

3.3. Deleting Application

This method deletes an application given the application ID.

Resource identifier	/app/{appId}
HTTP method	DELETE
Input Parameter	--
Response	--
Status code	200 if OK, otherwise the error code

3.4. Deleting applications

This method deletes all the existing applications.

Resource identifier	/app
HTTP method	DELETE
Input Parameter	--
Response	--
Status code	200 if OK, otherwise the error code

3.5. Getting application description

This method returns the application description in JSON format (Figure 2). Among others, application description has application ID which can be found with “id” key (e.g., id:12345) and application status which can be found with “status” key.


```
{
"name":"liquidiot-temperature",
"version":"1.0.0",
"description":"",
"main":"measure.js",
"scripts":{"test":"echo \"Error: no test specified\" && exit 1"},
"author":"",
```

"license":"ISC",
"**id**":12345,
"**status**":"initializing",
“interfaces”:”[canMeasureTemperature]”
}

Figure 2 – Application description

Each application has four statuses: *initializing*, *crashed*, *running*, *paused*. The application is in *initializing* status when it is being created. Then it goes to either *running* or *crashed* status. When the application is stopped, it goes to *paused* status.

In running status, the application code is either called in certain intervals (by Javascript timing events) or just once. This can be configured by application developer. When the application is in running mode, it exposes its APIs. The *paused* mode stops application from both running and exposing its APIs.

To get the application status, application ID (i.e. appId) must be specified.

Status	**meaning**
initializing	The application is being created.
running	The application code is called by Javascript timer events either in certain intervals or only once.
crashed	The application encountered an error.
paused	The application stops running.

Resource identifier	/app/{appId}
HTTP method	GET
Input Parameter	--
Response	Application description in JSON format.
Status code	200 if OK, otherwise the error code

3.6. Listing Applications

This method lists descriptions of the existing applications in JSON array format.

Resource identifier	/app
HTTP method	GET
Input Parameter	--
Response	A list of descriptions of the existing application (JSON Array)
Status code	200 if OK, otherwise the error code

3.7. Starting application

This method starts the application (given its ID). Actually, this method creates a timer object, which calls the application either in certain intervals or only once. As the input parameter, a JSON string that specifies the target mode (*running*) must be provided.

Resource identifier	/app/{appId}
HTTP method	PUT
Input Parameter	{status : "running"}
Response	Instance description with the changed "status".
Status code	200 if OK, otherwise the error code

3.8. Stopping application

This method stops the application (given its ID). If the application is configured to be called in certain intervals, this method deletes the timer object that is calling the application and in turn transforms the application to the *paused* mode. As the input parameter, a JSON string that specifies the target mode (*paused*) must be provided.

Resource identifier	/app/{appId}
HTTP method	PUT
Input Parameter	{status : "paused"}
Response	Instance description with the changed "status".
Status code	200 if OK, otherwise the error code

4. File structure of the Liquid-IoT application:

The Liquid-IoT application should have a certain file structure and contain some specific files:

a) *Package.json file*: It is the same package.json file of a typical node.js application. The content is in JSON format and it should have at least the *main* property which defines the entry point (main file) of the application.
b) *<MainFile>.js file*: The main file (the entry point) of the application. *package.json* file has a *name* property that specifies the name of the main file of the application.
c) *Resources folder:* The application may have a *resources* folder that contains the needed resources. For example, the application that plays a sound expects to get the audio file from this folder.
d) *agent.js file*: It is a superclass that provides functionality to both applications and LIoTRE.
e) *liquidiot*.json file: It specifies which interfaces the application implements (through *interface* property) and which devices it can be deployed (through *deviceCapability* property). The content is in JSON format.
f) The application may have other files than the main file. They should be in the root folder of the application.

The file structure of Liquid-IoT application is shown in Figure 3.

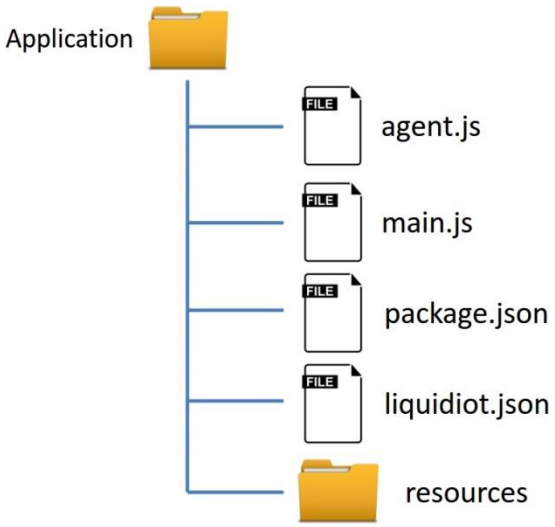

Figure 3 - Liquid-IoT application File structure