Programming IoT Gateways With macchina.io

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
Maria Elend 143
9182 Maria Elend
Austria
guenter.obiltschnig@appinf.com

This article shows how to program IoT Gateway devices in macchina.io, a new open source software platform.

IoT Gateways play a central role in many Internet of Things applications. They are the link between (wireless and low-power) sensor and fieldbus networks, as well as "legacy" devices (typically connected via RS-232, RS-485 or USB interfaces) on the one side, and cloud or enterprise systems on the other side. Typically, an IoT Gateway device is based on a powerful ARM Cortex or Intel SoC and running a Linux operating system. Some devices include high-speed cellular network interfaces. IoT Gateways are often used in highly customer-specific environments, so easy programmability is an important feature. The application-specific code running on these devices in many cases is neither very complex nor performance-critical. Typically, it has to acquire some data from connected sensors or devices, preprocess this data, and send it to a cloud or enterprise server for further processing and/or storage.

In the last few years JavaScript has become one of the most popular programming languages. While having its roots in client-side web development, JavaScript is now also a very popular programming language for server-side development, due to projects like node.js. Furthermore, the competition among the Firefox, Chrome and Safari browsers has brought with it a race for the fastest JavaScript engine. Most modern JavaScript engines like Google's V8 now compile JavaScript into machine code, resulting in good performance even for non-trivial applications, or on resource constrained systems.

On the other hand, C++ has also seen a renaissance in recent years, mostly driven by two factors: the new C++11 and C++14 standards, bringing modern features like lambdas to the language, and the need for maximum efficiency and performance, which, despite years of promises, are still not matched by virtual machine based languages like Java or the .NET family of languages.

A new open source platform, macchina.io, combines the flexibility of JavaScript for rapid application development with the power and performance of native C++ code. macchina.io is mostly implemented in C++, for maximum performance and efficiency. Although JavaScript plays a big role in macchina.io as the preferred language for high-level application development, it is not used much in the implementation of macchina.io itself, except for some parts of the web user interface. The combination of JavaScript for rapid high-level development, with C++ for

performance critical, or low-level code, makes macchina.io a perfect platform for IoT gateway devices. Furthermore, a unique bridging system and a code generator make it easy to consume services written in C++ from JavaScript, without the need to manually write awkward glue code required to make C++ objects accessible from JavaScript.

The foundation of macchina.io is the so-called "Platform" (see Figure 1). It consists of the POCO C++ Libraries, the Remoting framework, the Open Service Platform (OSP) and the JavaScript environment, based on the V8 JavaScript engine.

Figure 1: macchina.io Overview

The POCO C++ Libraries are modern, powerful open source C++ class libraries and frameworks for building network- and internet-based applications that run on desktop, server, mobile and embedded systems. They provide essential features such as platform abstraction, multithreading, XML and JSON processing, filesystem access, logging, stream, datagram and multicast sockets, HTTP server and client, SSL/TLS, etc. Virtually everything implemented in macchina.io (except some integrated third-party open source projects) is based on the POCO C++ Libraries.

The Open Service Platform (OSP) enables the creation, deployment and management of dynamically extensible, modular applications, based on a powerful plug-in and services model. Applications built with OSP can be extended, upgraded and managed even when deployed in the field. At the core of OSP lies a powerful software component (plug-in) and services model based on the concept of bundles. A bundle is a deployable entity, consisting of both executable code (shared libraries or JavaScript) and the required configuration, data and resource files (e.g., HTML documents, images and stylesheets required for a web site). Bundles extend the

functionality of an application by providing certain services. A central Service Registry allows bundles to discover services provided by other bundles. Bundles can be installed, upgraded, started, stopped or removed from an application (programmatically, or using a web- or console based administration utility) without the need to terminate and restart the application.

Remoting is a distributed objects and web services framework for C++. The framework enables distributed applications, implementing high-level object-based inter-process communication (IPC), remote method invocation (RMI) or web services based on SOAP/WSDL. In macchina.io, Remoting is used for the C++-to-JavaScript bridging mechanism. It can also be used for efficient RPC-based inter-process communication, using the TCP transport.

The JavaScript environment in macchina.io is based on the Google V8 engine. V8 is the JavaScript engine used in the Google Chrome browser and node.js, a well-known server-side JavaScript platform. V8 compiles JavaScript directly into optimized machine code, ensuring good performance. There are bindings that enable using certain features of the POCO C++ Libraries and OSP in JavaScript code. Examples are database access (SQLite), HTTP(S) client, access to application configuration and environment, OSP service registry, etc. JavaScript can also be used to write servlets and server pages for the built-in web server. This makes it easy to visualize sensor data on a web page hosted by macchina.io's built in web server.

The IoT Components are the "heart" of macchina.io. Various OSP bundles and services implement features such as interfaces to devices and sensors, network protocols such as MQTT, interfaces to cloud services (e.g., for sending SMS or Twitter messages), and the web-based user interface of macchina.io. All this is available to JavaScript and C++ code.

macchina.io defines generic interfaces for various kinds of sensors and devices. Based on these interfaces, different implementations are available that make specific sensors and devices available in macchina.io. There are interfaces and implementations for generic sensor types such as temperature or humidity sensors, GNSS/GPS receivers, accelerometers, triggers, GPIO ports, serial port devices, barcode readers, etc. Additional sensor types and implementations can be easily added as well.

macchina.io implements various protocols for talking to sensor networks, automation systems, or cloud services. One such protocol is MQTT, a publish-subscribe based "light weight" messaging protocol for use on top of the TCP/IP protocol, which is popular for building cloud-connected IoT applications.

The user interface of macchina.io is entirely web-based (Figure 2). Some parts of the web interface (e.g., System Information, Sensors and Devices, GNSS Tracking, MQTT Clients) are built entirely in JavaScript, both on the client and on the server side. Other parts combine JavaScript on the client side with C++ REST services on the server side. A highlight of the web user interface is the "Playground" app (Figure 3). It provides a comfortable browser-based JavaScript editor and allows running JavaScript code on the device. This allows for very easy prototyping and experimentation. Instead of having to compile code on a host system, then transfer the resulting binary to the device, the code can be edited directly on the device and run with the simple click of a button.

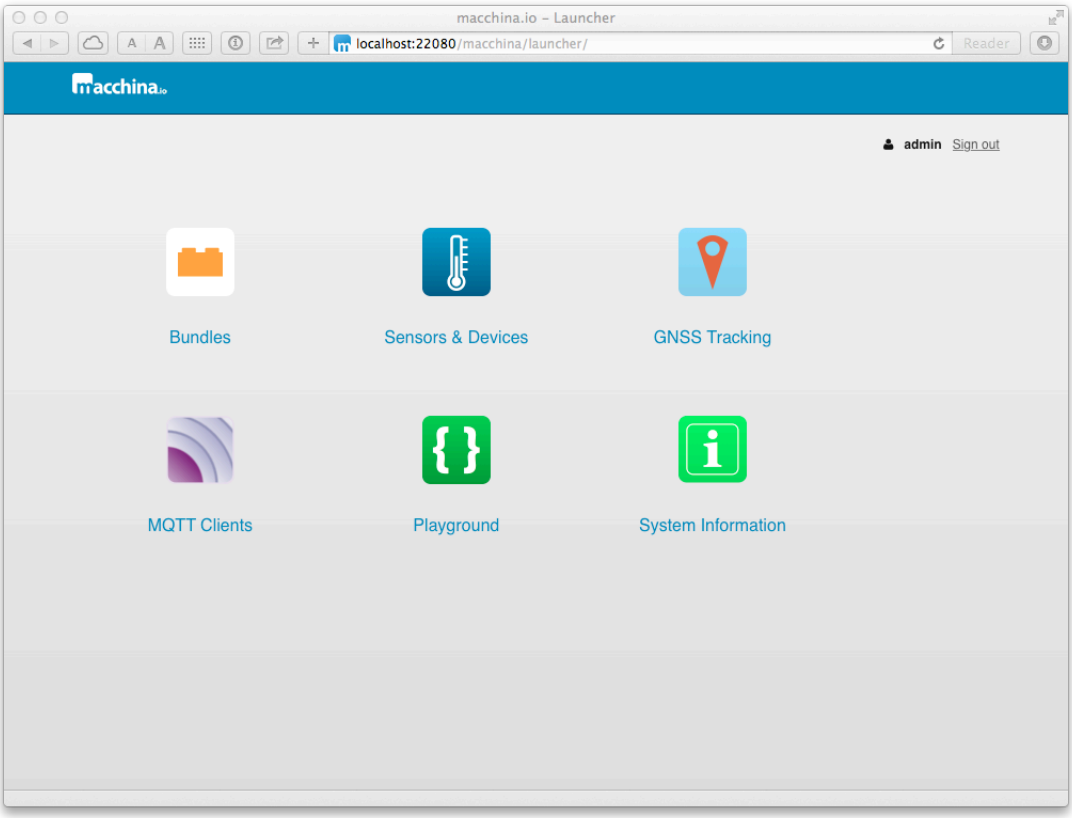

Figure 2: The macchina.io Web User Interface

The Playground comes pre-loaded with the macchina.io variant of the "Hello, world!" program - a short script that finds a temperature sensor and obtains the current temperature from the sensor. The script is shown in the following.

```
// Search for temperature sensors in the Service Registry
var temperatureRefs = serviceRegistry.find(
    'io.macchina.physicalQuantity == "temperature"');
if (temperatureRefs.length > 0)
{
    // Found at least one temperature sensor – resolve it.
    var temperatureSensor = temperatureRefs[0].instance();
    // Get current temperature.
    var temperature = temperatureSensor.value();
    logger.information('Current temperature: ' + temperature);
}
else
{
    logger.error('No temperature sensor found.');
}
```

Example 1: The macchina.io "Hello, world!" JavaScript program

The program shows two things: how to obtain a sensor service object from the Service Registry, and how to write logging output.

To find available sensors in the system, the macchina.io Service Registry is used. All available sensors and devices will be represented as service objects using that registry. In order to find a specific object, the service registry supports a simple query expression language that allows finding services based on their properties. In the example above, we specifically look for a temperature sensor. In macchina.io, sensors always have a property named *io.macchina.physicalQuantity* that can be

used to search for sensors that measure a specific physical quantity. The find method will return an array of service references. Service references are different from actual service objects. Their purpose is to store service properties (like the mentioned *io.macchina.physicalQuantity*), as well as a reference to the actual service object. The actual service object can be obtained by calling the *instance()* function, like we do in the sample. Once we have the temperature sensor object, we can use it to obtain the current value by calling the *value()* function.

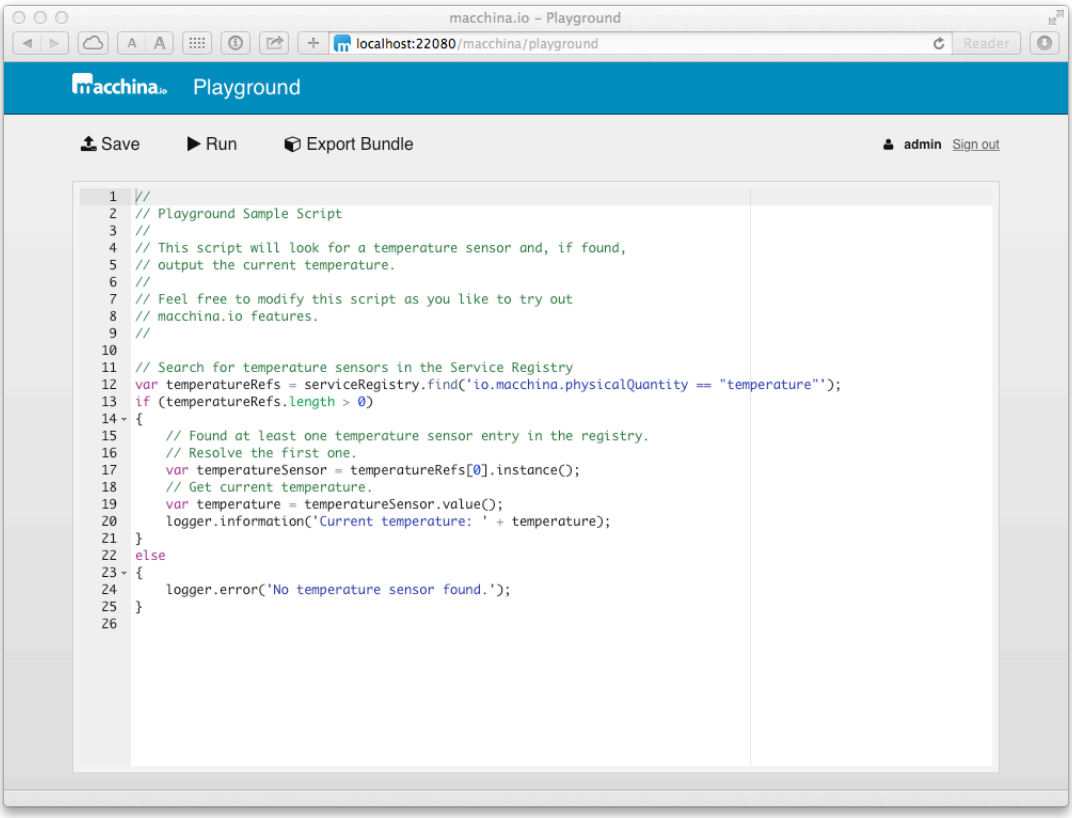

Figure 3: The macchina.io Playground App

In the Playground app, the script can be run on the device by clicking the "Run" button above the editor.

As a next step, the program can be modified so that it will periodically output the current temperature. This is done by setting up a timer, using the *setInterval()* function. Here's the modified code:

```
var temperatureRefs = serviceRegistry.find(
    'io.macchina.physicalQuantity == "temperature"');
if (temperatureRefs.length > 0)
{
    var temperatureSensor = temperatureRefs[0].instance();
    logger.information('Found temperature sensor.');
    setInterval(
        function() {
            logger.information('Current temperature: ' +
                temperatureSensor.value());
        },
        1000
    );
}
else
{
```

```
    logger.error('No temperature sensor found.');
}
```

Example 2: Periodically querying sensor data.

Sensor objects also support events, so scripts can be notified when a sensor measurement changes. A script can register a callback function to be notified as follows:

```
temperatureSensor.on('valueChanged',
    function(event) {
        logger.information('Temperature changed: ' +
            event.data);
    }
);
```

Example 3: Events and callback functions

JavaScript code can also invoke web services. The final code example shows how to invoke a HTTPS based web service from JavaScript. If the temperature exceeds a certain limit, we'll send a SMS message using the Twilio SMS cloud service. First, the JavaScript function to send a SMS message:

```
function sendSMS(message)
{
    var username = 'username';
    var password = 'password';
    var from = '+1234567890';
    var to = '+1432098765';

    var twilioHttpRequest = new HTTPRequest(
        'POST',
        'https://api.twilio.com/2010-04-01/Accounts/' +
        username + '/SMS/Messages');
    twilioHttpRequest.authenticate(username, password);
    twilioHttpRequest.contentType =
        'application/x-www-form-urlencoded';
    twilioHttpRequest.content =
        'From=' + encodeURIComponent(from) +
        '&To=' + encodeURIComponent(to) +
        '&Body=' + encodeURIComponent(message);

    var response = twilioHttpRequest.send(function(result) {
        logger.information('SMS sent with HTTP status: ' +
            result.response.status);
        logger.information(result.response.content);
    });
}
```

Example 4: Sending a SMS message using the Twilio cloud service

And finally the code to check the temperature, and to send the message if a temperature limit (30 ˚C) is exceeded:

```
var smsSent = false;

var temperatureRefs = serviceRegistry.find(
    'io.macchina.physicalQuantity == "temperature"');
if (temperatureRefs.length > 0)
{
    var temperatureSensor = temperatureRefs[0].instance();
    temperatureSensor.on('valueChanged',
        function(event) {
            if (event.data > 30 && !smsSent)
            {
```

```
                sendSMS('Temperature limit exceeded!');
                smsSent = true;
            }
        }
    );
}
```

Example 5: Monitoring a temperature and alerting via SMS

In this article we have introduced macchina.io, a new open source toolkit for programming IoT Gateway devices, combining JavaScript for rapid high-level application development with C++ for high performance low-level development. Beside being a great platform for manufacturers of IoT Gateway devices, macchina.io is also great for prototyping, e.g. using a Raspberry Pi combined with Tinkerforge. macchina.io is licensed under the Apache 2.0 license and available from GitHub. More information and downloads can be found at http://macchina.io.