Using the MEAN Stack to Implement a RESTful Service for an Internet of Things Application

Andrew John Poulter*, Steven J. Johnston†, Simon J. Cox‡
University of Southampton,
Faculty of Engineering and the Environment,
Southampton, United Kingdom
*Email: a.j.poulter@soton.ac.uk** *sjj698@zepler.org*† *s.j.cox@soton.ac.uk*‡

***Abstract*—This paper examines the components of the MEAN development stack (MongoDb, Express.js, Angular.js, & Node.js), and demonstrate their benefits and appropriateness to be used in implementing RESTful web-service APIs for Internet of Things (IoT) appliances. In particular, we show an end-to-end example of this stack and discuss in detail the various components required. The paper also describes an approach to establishing a secure mechanism for communicating with IoT devices, using pull-communications.**

***Keywords*-Internet of Things; IoT; REST; MEAN; web programming; MongoDb, Express.js, Angular.js, Node.js**

I. Introduction

The phrase *Internet of Things* (IoT), describes the fusion of electronic hardware devices, and Internet connectivity. Although in its strictest sense this term refers to a heterogeneous network of interconnected devices: in practice many IoT devices are not directly connected to each other — but rather connect to other services which then provide the interfaces to other devices, or to human users. Such devices are increasingly becoming common-place in homes, offices, and in industry: for example *smart* thermostats or lighting controllers. This paper examines the use of the *MEAN* web development tool stack (named for its constituent parts: MongoDb, Express.js, Angular.js, & Node.js) for developing back-end services and front-end web-based user interfaces for the data produced by such IoT devices.

II. The MEAN Stack

Developers of dynamic web applications have been using the *LAMP* open-source tool stack [1] (consisting of the Linux Operating System, the Apache Web Server, MySQL as a database and PHP as the scripting language) for some time. However, a new tool stack for web-application development has emerged over the last few years — known as the *MEAN Stack* or just MEAN.

MEAN takes it's names from the four tools that together provide both client & server-side components for interactive web applications: MongoDb which provides the object-database; Express.js which provides a framework for web routing; Angular.js for web applications; and Node.js — the JavaScript engine, and web server component [2].

All four of these tools are based around the JavaScript language — which although initially developed for client-side web programming has entered into common usage for server-side programming, thanks in large part to environments such as Node.js.

A. Node.js

Although canonically listed last when referring to MEAN, Node.js (or just *Node*) is the most important tool of the stack. Built around Google's *V8* JavaScript engine (originally written to execute client-side JavaScript, within the Chrome web browser), and implemented in C++; Node provides a high-performance, asynchronous event-based server [3]. Node can be used to build a lightweight and high-performance web server environment [4], ideal for constructing web-service APIs. It is used for this purpose by a number of major companies — including Walmart [5] & PayPal [6].

B. Express.js

Express.js builds on the underlying capability of Node, by providing a web application server framework. This framework provides a wrapper around a lower-level Node interface: giving the developer, a convenient means to handle routing and HTTP operations (such as `GET` and `POST`). Express.js facilitates a simplified and more elegant solution than (re-)implementing these services directly using Node.

C. MongoDb

The majority of web-services will require some sort of storage: often in the form of a database management system. Whilst traditionally that might have been provided using an SQL-based Relational Database Management System (such as MySQL or SQLServer) there is a growing trend to use a NoSQL type of database. NoSQL (also known as "No Only SQL") databases can be used to provide a more flexible "document-oriented database" with a dynamic schema.

MongoDb is a high-performance NoSQL database built around the JSON data format [7] — and as such is ideally suited to a server-side JavaScript environments such as those provided by Node. In October 2015, MongoDb was the the most popular document-oriented NoSQL database, and

978-1-5090-0366-2/15/$31.00 ©2015 IEEE

4th most popular Database Management System overall: as measured by the *"Knowledge Base of Relational and NoSQL Database Management Systems"* [8].

D. Angular.js

The last part of the MEAN stack is Angular.js (or *Angular*). Angular is an open-source web application framework, maintained by Google, which provides a client-side framework for MVC (Model-View-Controller) [9] single page web applications.

To gain the maximum benefit from Angular, it can be combined with two other packages: *Yeoman* & *Bootstrap*.

Yeoman provides an environment which enables the use of *generators*: simple script-based tools that can be used to scaffold the bare-bones of a Angular web app. The Yeoman project teams describe it as a tool which "...can help developers quickly build beautiful web applications" [10].

Bootstrap is a popular Open-Source CSS framework (originally created by Twitter), which is described as "...the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web". [11] Bootstrap provides elegantly designed CSS elements, making it easy to design web content with a clean, modern look.

Together, the combination of these tools with the underlying logic implemented with Angular, make it very easy to create a powerful and richly designed web application, which can consume the web-services provided by the other tools in the stack.

III. Representational State Transfer & the Internet of Things

Representational State Transfer [12], widely known as the REST or RESTful model for web-services, uses the native HTTP operations: `POST`, `GET`, `PUT` & `DELETE` [13] to map on to the four fundamental database operations — Create, Read, Update & Delete. A simple API can be built to link these four HTTP *verbs* to functions which Create, Read, Delete or Update records within a web-service. This service can then be consumed by any type of authenticated client device.

The client can be a web page, or web application; or the service can be programatically consumed by a custom-written software application on a PC, smart phone, tablet or other device; from within a simple script (for example in Python); or a highly-complex data-science application implemented in a low-level language, or a tool such as MATLAB or R.

It is clear that a RESTful API maps very well onto many classes of IoT application. For example, an IoT appliance that has no in-built User Interface can act as a special-purpose posting-client: accessing the the API, it can *push* its sensor data to a central server using a `POST` (Create) method. The data thus produced is centrally collated and is available to be consumed by `GET`-ing the data from the service.

IV. The advantages of the MEAN stack for implementing an IoT web-service

A. Node

Whilst there are a multitude of ways to implement a REST API, doing so using the MEAN stack, and specifically the use of Nodes & Express, enhances productivity by reducing the development effort required; whilst delivering an effective, efficient, and highly-scalable implementation. Although presently still less commonly known than the LAMP stack, MEAN appears likely to become influential; especially given Microsoft's increasing involvement with Node (including creating their own version of Node for ARM processors) [14] as a part of *Windows 10 IoT Core* — which runs on ARM devices such as the Raspberry Pi 2 [15].

Research has shown [4] that a Node server significantly outperforms both Apache and Nginx [16] for serving dynamic content — and Node's implementation of JavaScript is more than 2.5 times faster than the more traditional PHP approach, by efficiently utilizing available hardware.

B. JSON

A REST API needs to consume and produce data, encoded in a consistent manner. Whilst many choices exist (XML, YAML, etc.) a popular choice is JSON. JSON is well suited to data that needs to be both human and machine-readable. JSON data is, arguably, easier for a human reader to understand than XML; and can also be easily parsed by a computer. Tools and libraries to parse JSON exist in all major programming languages and environments.

JSON's *key-value pair* format is ideal for use with regular parametric data — such as may be produced by a sensor device, or transmitted as a command-and-control message to a device or actuator.

From Node or another JavaScript implementation JSON data is already in the native format & as such requires no further parsing unlike, for example, XML.

C. MongoDb

The advantage of the use of a document-oriented database, such as MongoDb is that is offers a dynamic (rather than fixed, and rigid) schema [17]. This means that if the data structure of the database is required to change, as a result of needing to store additional parameters; new data with a different structure can be accommodated within the database, alongside earlier data, without the need to perform large-scale data manipulation on the whole database.

MongoDb is also a good choice for storing JSON encoded data. MongoDb internally stores data in an efficient binary JSON format (BSON) — which allows for quick and easy

import and export. It is also is also ideally suited to storing and processing large volumes of data. It can scale to thousands of nodes and petabytes of data [17].

MongoDb also exhibits better runtime performance for simple operations at a small-scale (single node), than Microsoft SQL Server Express [18].

V. Implementing a REST API for an IoT device using MEAN

A simple example of a REST IoT *back-end* service was constructed using MEAN, in order to further explore its utility for such an application.

A. The Node & Express web-service

An example web-service was built using Node & Express, with the corresponding data stored on a MongoDb database running on the same server. As the server was built to be an experimental platform, it provided a full implementation of a REST API — permitting four operations (`POST`, `GET`, `PUT` & `DELETE`) on all data. In a real implementation (especially one where there was a requirement for data integrity) it would be good practice not to support `PUT` operations — rendering the data immutable. Similarly, providing a `DELETE` operation for individual or bulk data may be undesirable.

The initial implementation did not use any type of authentication, and permitted all operations to be carried out by any connected user or device. Whilst this is a valid approach for an experimental server operating on a private network, this is clearly not a model that should be adopted for any publicly addressable, Internet connected service. Hosting the service on a server using HTTPS, and modifying the service such that some or all API calls require authentication via Basic authentication [19] is very simple however, and was explored in subsequent work.

B. Hardware: a simple IoT device

Once the service had been implemented, an example of a sensor device was constructed from commodity development hardware. The device consisted of a Microchip MCP9808 temperature sensor, which was polled at a pre-determined time interval over an I2C interface by code running on an Ateml ATMEGA328 microcontroller. Together the microcontroller and the temperature sensor create a "sensor unit": which transmitted the temperature, as an ASCII string, over a wired serial link to a BeagleBone Black single-board computer. The BeagleBone Black then combined the raw data, with a *device ID* and a timestamp, before `POST`-ing the data to the web-service using a Python script, written using the popular *Requests* library [20].

The use of a dedicated microcontroller to act as an interface to the lower-level hardware meant that the sensor hardware becomes genericized, allowing the microcontroller to provide a consistent interface to the gateway portion of the device, enabling hardware to be substituted without significant modification to the design of the device.

This modular approach also gave maximum flexibility to the overall design of the device. The BeagleBone acted as a gateway — and can be trivially modified to handle multiple sensor units. Additionally the wired serial link can easily be reimplemented to use a wireless link — such as an XBee ZigBee module — if the installation calls for the location of the sensor to be remote from the gateway.

The use of low-powered, modularlized hardware communicating with more powerful *base-station* gateways means that the low-cost modules can be deployed widely without being constrained by connectivity or power.

The MEAN web-service recorded the data generated by this hardware setup as four JSON key-value pairs (the device identifier, the timestamp, the temperature recorded, and the MongoDb unique ID for the item). Data can be retrieved from the service either by accessing specific data items (by ID) or by retrieving all of the data. In either case data was returned as JSON. Post-processing this data, for example plotting data over time, can then be conducted using any tool or software capable of accessing JSON data via a web API.

C. An Angular web application

The web-service was also used to provide the back-end to a single page web application, implemented using the combination of Angular.js, Yeoman, and Bootstrap described in Section II-D.

This web application provided a means to view the raw data for a given device ID, as well as providing a means to delete and edit individual data points. Additional functionality, such as specific visualization of the data, the use of an authenticated API & secure data links, and functionality to permit configuration of the device can also be added.

D. The use of Node.js within a device

During this work, we explored running JavaScript code via Node.js directly on a device. There are a number of open-source libraries that enable high-level scripting languages such as Python and JavaScript, to directly interface with the hardware on a device. One such library, is the Node library *onoff* [21], which enables direct access to GPIO pins from JavaScript, on a number of supported development hardware platforms: including Raspberry Pi and BeagleBone. Other libraries support access to hardware data busses: for example I2C and SPI.

With the increase in availability of single-board computers (and the significant decrease in their cost) seen in recent years, and the significant computing power that they are now able to provide: there is a reduced requirement to tightly optimize embedded code running on these devices, in comparison to the situation of even a few years ago. This additional computing power (available at low-cost thanks

to the economies of scale resulting from the smartphone revolution) has meant that embedded systems are now able to benefit from greater abstraction between the hardware and the software.

Although there are some productivity benefits from developing software using scripting languages — it is not yet clear how well these translate to their use for software or firmware to be run on devices. Whilst these types of language may be well suited to some activities (such as early prototyping, and hobbyist use) there are some performance overheads to using these languages — and (critically for real-world applications) a significant configuration control overhead, when compared to statically linked code. Code written using one version of a library or module may not work with another version and versions of libraries are often tied to specific versions of the underlaying language interpreter.

As such, although there may be some merit to the use of scripting languages to implement firmware on a device — further work is required to more formally assess the benefits and costs; and conduct a more rigorous examination of any performance and security issues that may be present.

VI. A more advanced implementation of an IoT service — using the MEAN stack

Whilst the use-case presented in Section III and explored in Section V is ideal for simple cases; this kind of *write-only* connection from the device, to the Internet, is only appropriate for a relatively small number of applications. Although such a model enables an Internet connected sensor to push data to a central repository — it does not provide a means for an Internet-connected *actuator* to receive *command-and-control* data (such as would be required to turn on a light, or change the temperature at which a system operates). Even a *sensor-only* device may have a requirement for a return path — for example, to modify sensor parameters (such as frequency of sensing) or to provide a means to perform a software, or firmware, update to the device.

One approach to implement this inbound communications, would be to open a direct connection from the end-user (or their client application) to the device; however this poses risks for the security of the device (and thus the system) by potentially exposing the device to malicious or erroneously malformed communications messages from the user. It also places a requirement on the device to authenticate an inbound communication; and to remain in a state where it is ready to receive communications.

A. Server-based pull-communications

An alternative is to provide a reverse path for communications back to the device, by having the device periodically (at a frequency appropriate to the application) poll the web-service, looking for updated command-and-control messages. This type of *pull-communications* to the device is inherently more secure, as it obviates the need for a direct connection into the device from a remote user. By managing all inbound connections at the server, it also makes it easier to securely and definitively authenticate the source of the command-and-control messages received. Providing the device can securely establish the identity of the web-service: using security techniques common-place within web applications, such as Transport Layer Security (TLS) [22], and providing that the device (or the user or user client) is securely authenticated to the server: the device itself never has to consider the security challenges of directly authenticating a user or other command-agent. This authentication can use either HTTP Basic authentication, or a more advanced protocol.

It also potentially permits a lower power-consumption from the device as it may enter a sleep state during the periods between the predetermined communications intervals rather than remaining constantly in a higher-power state, in order to receive communications at unknown times.

Using this paradigm for pull-communications, an IoT device can collect and then act upon messages: depending on their content. The messages may be a simple command (e.g. *"turn on light #0A49"*), or may be more a more complex control message (e.g. *"adjust heater output to maintain a room temperature of 20.5°C"*).

B. Remote software updates

Using this *notification* approach, system control messages can also be specified: for example to instruct the device to retrieve a software update from the server. This paradigm is analogous to the way that Operating System patches are often distributed: remote systems do not push the update to the computer — but rather the computer monitors a known *repository*, waiting for an indication that updates are available. The use of this technique also means that the computer does not have to be directly addressable from the Internet: as may be the case if it is connected to a network which is protected by a Firewall or behind a router using Network Address Translation (NAT). When updates are available, these are then downloaded by the computer. The origin of these updates may be verified using TLS, and their content checked for integrity using a hashing function such as SHA-1.

Software updates for an IoT device can range from security patches and simple updates to the firmware — all the way through to a significant reconfiguration of the operation of the device. Such reconfiguration can modify the device to provide new functionality — and can utilize previously dormant hardware within the device. For example, a remote sensor could be reconfigured to operate more effectively, based on analysis of data that it has previously gathered — information that was not known when the device was originally deployed. The ability to remotely reconfigure a device that has been deployed is very appealing: although there are a number of ethical considerations to so doing.

To provide additional security to the process, cryptographic algorithms can be used to sign the software and messages as authentically originating from an approved source — and by utilizing a private key within the device, to ensure that only messages, data or software updates intended intended for a specific device are usable by that device.

C. Alternatives to Basic Authentication — OAuth2

The OAuth2 protocol [23] is an open standard protocol designed to negotiate access to a resource, and facilitate authentication via the use of access tokens. Although most commonly used to provide a mechanism to authenticate a user to a website, using identity services from a third-party *provider* (for example: Google), without exposing the user's credentials from that provider to the site they wish to use; the protocol also supports authenticating a user to a resource, where the site hosting the resource is its own identity provider.

The advantage of doing this (versus a more simplistic approach, directly using the user credentials) is that the password never has to be stored by the client. The finer granularity this method offers means that a user can revoke access for a given token (for example, a particular client application), without effecting other tokens.

The OAuth2 protocol defines four types of *Authorization Grant* — which can be used to negotiate access, and issue a token. These different grant types are each best suited for different applications. Although the four grant types describe a different negotiation process, upon successful completion they all result in the issue of an access token — which is used to authenticate the connection by being sent as a part of the HTTP header (hence it still requires the use of TLS). An OAuth2 service can, depending on the implementation, also issue a *refresh token* to facilitate reissuance of an access token, after the original token has expired.

The *Authorization Code* grant, and the *Implicit* grant are primarily intended to enable web-services to access data held on another web-service. They can be used to authenticate a user of a web application associated with an IoT device against the API of another web resource — such as Twitter or Facebook. For example, a user could choose to use the IoT device's web application share data from that device — in the form of a tweet, or Facebook message.

The *Resource Owner Credentials* grant is designed to give access to a (semi-trusted) client application. The user needs to provide their password at the time of first authentication (or any subsequent re-authentication in the event of token expiry or revocation) — but this is never stored by the client application. In this use-case a user can have a client application (whether on a mobile device, or a more traditional computer) to access the web-service: using a Resource Owner Credentials access grant to obtain access the web-service.

Lastly, the *Client Credentials* grant type is exclusively used for machine-to-machine communications where there is no specific human user involved: rather the device or client authenticates itself (as a known and trusted client) with the service. This uses a very similar process to the Resource Owner grant — but using credentials relating to the client itself, rather than a human user.

Both the Resource Owner, and Client Credentials grants are well suited for use in the context of IoT; and the server-side element of both can easily be implemented using Node, Express, and other libraries (such as OAuth2orize [24] & Passport [25]).

An implementation of a web-service using an OAuth2 Resource Owner Credentials grant was constructed during follow-on work to the original activity.

D. Containerization

One of the major trends of the last year has been the increasing interest in, and usage of, *containerization* to deploy applications — using tools such as *Docker* [26]. Docker enables a developer to package an application, together with all of its dependencies, into a standardized and lightweight *container* which can then be deployed to a production environment. This technique makes it possible for a developer to ensure that all of the required dependencies will be met, regardless of the state of the configuration of the base operating system on the machine onto which the application is to be deployed.

Whilst there is certainly potential to utilize containerization services to facilitate a MEAN web-service or application, there are also other (potentially lighter-weight) approaches that can be adopted. Node provides the *node package manager* (npm) [27] — which uses a JSON file to identify the configuration and dependencies of a Node application; and can automatically download all required library code. Because of the level of abstraction from the underlying OS that Node provides: this technique can be used to enable npm to automatically configure an environment to run any Node application. Similar package management tools exist in other scripting languages. This technique obviates, to some extent, the need for a containerized approach when using Node applications. However in a production environment there is still a necessity to ensure that lower-level dependencies (such as the correct versions of Node and MongoDb) are present. Containerization such as Docker provides one way to ensure that these can be rapidly deployed: and if such a method is being utilized then it can easily be extended to incorporate the Node application and its dependencies.

Whilst beyond the scope of this work, there is also the potential to use this containerization paradigm as a part of a process that can deliver atomic transactional updates to an IoT device's software or firmware. Further work is planned to investigate the validity of this approach.

VII. Conclusions

This work has shown that the MEAN stack is well suited to creating back-end services for the class of Internet of Things devices requiring a web-services API. MEAN offers productivity and performance advantages over more traditional tool stacks — such as LAMP.

A combination of Node & Express.js, with a MongoDb database provide an excellent implementation of a JSON-based web-service; not least because all of the tools within the stack will natively utilise the underlaying JSON data. Such a MEAN stack web-service is also highly scalable — thanks to the benefits of both Node and MongoDb.

Whilst formally assessing the merits of running Node and other scripting languages & interpreters directly on the hardware was beyond the scope of this work; this work has demonstrated that this is possible — and that it merits some further exploration and consideration of use-cases.

Using a web-service to securely enable remote software updates is both possible and appealing. Experimental work to implement and asses this for IoT devices is ongoing.

The use of software containers to facilitate the scalable deployment of MEAN stack web-service also merits further exploration as the technology matures.

References

[1] G. Lawton, "LAMP lights enterprise development efforts," *Computer*, vol. 38, no. 9, pp. 18–20, Sep. 2005. [Online]. Available: http://dx.doi.org/10.1109/MC.2005.304

[2] MEAN.io. (2015) MEAN — Full-Stack JavaScript Using MongoDB, Express, AngularJS, and Node.js. [Online]. Available: http://mean.io/

[3] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MIC.2010.145

[4] I. K. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Is Node.js a viable option for building modern web applications? A performance evaluation study," *Computing*, pp. 1–22, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00607-014-0394-9

[5] J. O'Dell, "Why Walmart is using Node.js," 2012. [Online]. Available: http://venturebeat.com/2012/01/24/why-walmart-is-using-node-js/

[6] Github. (2015) Projects, Applications, and Companies Using Node. [Online]. Available: https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node

[7] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format Interchange Format," 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159

[8] Knowledge Base of Relational and NoSQL Database Management Systems. [Online]. Available: http://db-engines.com/en/ranking

[9] A. Leff and J. T. Rayfield, "Web-application development using the Model/View/Controller design pattern," *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pp. 118–127, 2001.

[10] Yeoman: The web's scaffolding tools for modern webapps. [Online]. Available: http://yeoman.io

[11] Get Bootstrap. [Online]. Available: http://getbootstrap.com

[12] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 407–416. [Online]. Available: http://doi.acm.org/10.1145/337180.337228

[13] R. Fielding and J. Reschke. (2014) Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. [Online]. Available: https://tools.ietf.org/html/rfc7231

[14] J. Martin. (2015) Microsoft Forks Node.js to Support ARM. [Online]. Available: http://www.infoq.com/news/2015/05/nodejs-arm

[15] L. Upton. (2015, 30/4/2015) Windows 10 for iot. [Online]. Available: https://www.raspberrypi.org/blog/windows-10-for-iot/

[16] Welcome to NGINX Wiki's documentation. [Online]. Available: https://www.nginx.com/resources/wiki/

[17] Internet of Things : MongoDB. [Online]. Available: https://www.mongodb.com/use-cases/internet-of-things

[18] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL DB," *Proceedings of the 51st ACM Southeast Conference on - ACMSE '13*, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2498328.2500047

[19] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication," 2014. [Online]. Available: http://tools.ietf.org/html/rfc7235

[20] K. Reitz. Requests: HTTP for Humans. [Online]. Available: http://docs.python-requests.org/en/latest/

[21] OnOff – GPIO access and interrupt detection with JavaScript. [Online]. Available: https://github.com/fivdi/onoff

[22] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246

[23] D. Hardt, "The OAuth 2.0 Authorization Framework," 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749

[24] J. Hanson. OAuth2orize library. [Online]. Available: https://github.com/jaredhanson/oauth2orize

[25] ——. Passport : Simple, unobtrusive authentication for Node.js. [Online]. Available: http://passportjs.org

[26] What is Docker? [Online]. Available: https://www.docker.com/what-docker

[27] npm: the Node package manager. [Online]. Available: https://www.npmjs.com