Practical Test & Measurement

REST for Automated Test and Measurement

This article makes the case for using web interfaces and embracing web standards for automated testing of audio equipment, fundamentally exploring the possibilities of REST—which stands for Representational State Transfer—a widely accepted set of guidelines for creating reliable web APIs.

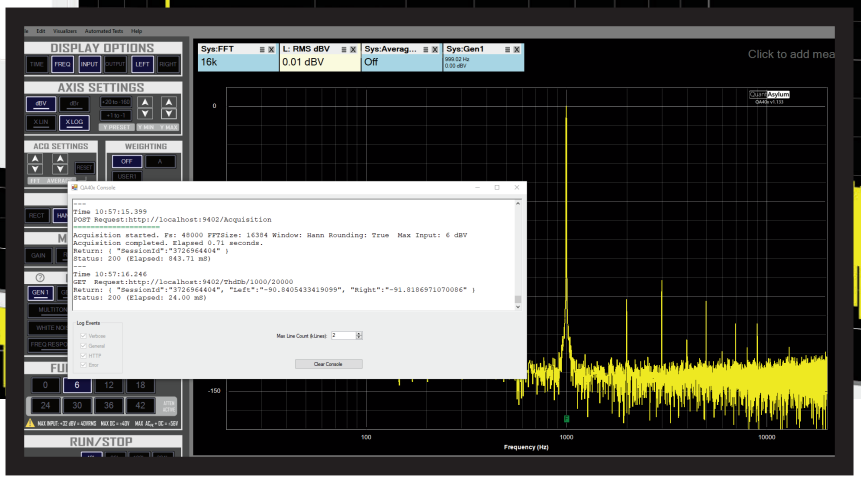

Photo 1: This is the QuantAsylum QA402 audio analyzer application, with a debug console on top, allowing you to see the processing of each HTTP request as the requests come across from the Jupyter application (or any application you might write). This makes debugging automation straightforward.

By
Matt Taylor

Over the decades, test and measurement solutions for manufacturing have evolved. HP Interface Bus (HP-IB) was born in the late 1960s to allow test instruments and computers to be connected together. The standard later became IEEE 488, often known as GPIB. And while the dominant physical interface has changed (USB and Ethernet are common today), the concept has largely remained the same with a widely used Application Programming Interface (API) eventually emerging known as VISA, for Virtual Instrument Software Architecture.

At the core of the VISA API is a driver for each instrument. The drivers handle mapping the strongly-typed test API to what is usually a loosely-typed text-based Standard Commands for Programmable Instruments (SCPI) control language that is sent over the physical layer.

And while the solution has worked well for the last 30 years, a concurrent but much larger revolution has taken place with a very similar problem statement: What is the best way for machines to talk to other machines in a generic sense? That is, how does a low-energy Bluetooth module communicate with a wearable computer? How does an app on your phone retrieve stock quotes from the Internet? How do you remotely set the state of a light switch in your home? These are all examples of Machine-to-Machine (M2M) communication.

In a generic sense, testing a product is no different than orchestrating an afternoon workout on the treadmill. The treadmill is reporting incline and speed, your watch is measuring heart rate, pulse oximetry, and temperature. Your phone aggregates that information, potentially adjusts the speed or incline, encrypts it, and uploads it to the cloud. Then it prepares a graph for you to view in real time as you work out. Later, you can go to the cloud via a web page, and do deep analysis on the data you've collected. From there, the sky is the limit on what can be done with the data (**Photo 1**).

What is important to understand are the pieces used to enable this seamless communication between so many disparate devices. Once you understand these elements, you'll see Test and Measurement in an entirely new light.

What Is REST?

REST stands for Representational State Transfer. REST isn't standard. It's an architectural prescription for how the Internet is glued together. At the core are requirements for security, performance and scalability, uniform interfaces, and the ability to evolve clients and servers separately. When you retrieve a web page, you are following the REST recipe, often known as being RESTful. This means there's a client and there's a server. Data on each server is exposed through a URL, and that data might change several times per second. For example, the URL:

```
http://localhost/temperature
```

will connect to your local machine and query the temperature. And every time you call that URL, the value returned might be different. If you wanted to know the temperature on another machine, you might call:

```
http://myWatch/temperature
```

or

```
http://myThermostat/humidity
```

If down the road the server API needs to change for some reason, you can add a new API by simply extending the URL:

```
http://myWatch/V1/temperature
```

Legacy systems can call the old URL, and new systems can call the new URL.

The URLs I mentioned will rely on an HTTP GET method to achieve the read. But how do you add new data to a remote machine? That is handled by an `HTTP POST` or `HTTP PUT`. Just as when you fill out a form to purchase something on the Internet, the data is encapsulated in a well-formed data interchange format, such as XML or JSON, and then encrypted for safe transport.

In short, the last few decades have delivered a framework for M2M communication that is nothing short of remarkable. It has proven extensible, secure, robust, scalable, and performant.

The Benefits of REST

So, given this framework for M2M communication, how can we apply it to test and measurement? Let's look at the various pieces that come into play.

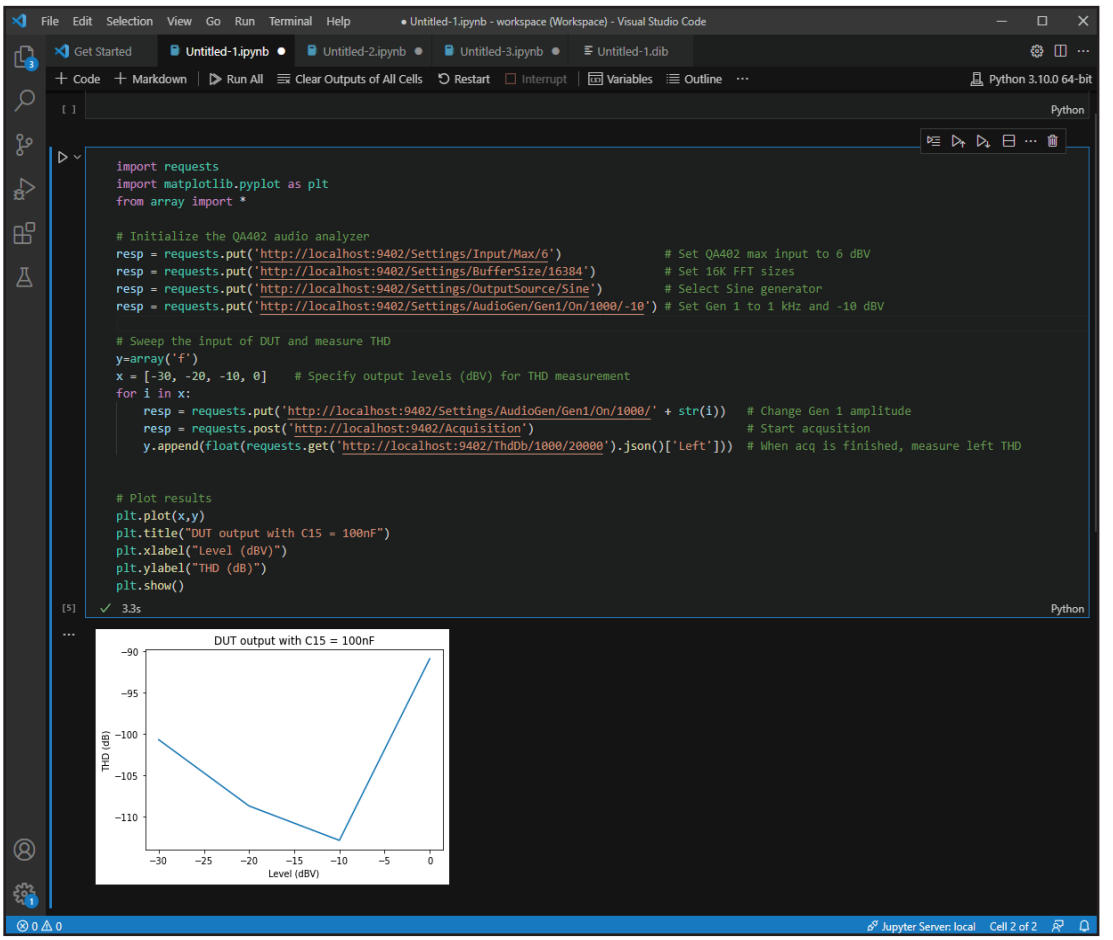

Photo 2: This image shows a Jupyter notebook using Python and direct HTTP calls to sweep and graph the THD measurement of a device under test (DUT). The calls are made using direct HTTP calls for illustration. Jupyter allows code, plots and rich text to live side-by-side, and is an excellent way to document the engineering evolution of a product during development. Jupyter is a common "front-end" for cloud computing and collaboration. At the bottom of the Jupyter code window, you can see it took 3.3 seconds to measure THD at four points using 16K FFTs.

Tools

As the M2M revolution has unfolded, a large set of overlapping tools have emerged to help cope with the challenges associated with M2M communication. At the center of it all is "big-data," data science, and the need to process extraordinary amounts of data pulled from a range of disparate locations. While the amount of data generated during test and measurement are small by big-data standards, we can benefit enormously from the analysis and scalability built into the various toolsets.

Free (and open-source) tools (e.g., Jupyter) permit an interactive environment where code, Markdown, and graphs can all live together (**Photo 2**). During development, this makes it easy to do something, such as plot THDN vs. frequency, change a capacitor, and run the THDN plot again, all while annotating your development journey in a rich-text markdown language.

Debugging tools (e.g., cURL) make it easy to create and debug complex M2M sequences if you are developing factory tests. And legacy tools (e.g., Matlab and LabVIEW) know how to speak REST already.

In short, the tools, ranging from development to deployment, are mature and pervasive, and often free. This becomes particularly important as manufacturing must scale.

Developers

Developers that understand test and measurement (GPIP, the interfaces, and the specific tools) are rare. It's a specialized skill that doesn't have much applicability outside of the factory realm. But M2M communication via REST is widely understood. Whether you are querying inventory from a supplier or handling credit-card processing, it's usually done through REST principals. And most developers understand it

About the Author

Matt Taylor holds a BSEE and is the president of QuantAsylum USA LLC, an audio test and measurement provider. Prior to QuantAsylum, he co-founded the Taiwanese electric scooter company Gogoro where he served as CTO. Before that, he worked at Microsoft, Motorola Cellular, and HTC. A lifelong musician, he fell in love with audio when he first entered a recording studio in the 1980s.

very well. Asking a developer to query data from a remote source via HTTP is a trivial task.

Languages and Environments

Languages such as Java, C#, and Python have also evolved. Making an `HTTP GET` or `PUT` or `POST` from a modern language is straightforward. There are no libraries needed, no new interfaces to learn, and no drivers. The operation, regardless of the underlying operating system is the same. On the QA402 Audio Analyzer, for example, if you want to know the THD of the last acquisition you made, with fundamental at 1kHz and a 20kHz bandwidth, you can type the following in your browser (or call from a program):

```
http://localhost:9402/ThdDb/1000/20000
```

And a JSON document is returned:

```
{
“SessionId”:”3307217056”,
“Left”:”-91.7134518113503”,
“Right”:”-94.3172168407987”
}
```

A JSON document is simply a collection of attribute-value pairs. We can see the left channel distortion was -91.73dB.

To request the distortion reading in Python would appear as follows. This line below is sending the HTTP request, and then asking for the value of the “Left” attribute in the returned JSON document. That value is a string, which is then converted to a float. Again, this is using standard language features. No drivers, no DLLs:

```
ThdDbLeft =
float(requests.get(‘http://
localhost:9402/ThdDb/1000/20000’).
json()[‘L eft’])
```

Of course, you’d want to abstract the calls shown above to give a layer of type-safety to the operation.

Performance and Scalability

An `HTTP GET` as shown for THD takes just a few milliseconds to complete. Even for large data structures the performance is very good. Requesting acquisition spectrum data (64K left channel and 64K right channel = 128K 64-bit doubles) data takes just 40mS or so. That’s largely because performance and scalability have driven the design of the Internet from the beginning, and languages such as Java and C# have followed suit. But it doesn’t stop there, as these languages have pushed further into advanced concepts such as asynchronous programming, which is a type of coding where the developer gets the performance gains associated with asynchronous calls without having to worry about the code overhead of callbacks. Asynchronous programming also makes it easy to parallelize a range of tasks. Modern programming languages have treated parallelization as a very important task, and as such there is strong language support for these important architectural elements.

In short, you can submit a range of measurement requests via HTTP before a single request has completed, and process them as they become available. And this can be done with code that is easy to write using modern languages.

Portability

With some planning, modern tools can allow you to seamlessly move a test environment from, say, Windows to Linux. You can develop in the lab on Windows or Mac and deploy to the manufacturing line onto low- cost Linux machines. Cross-platform applications are becoming increasingly common, and REST helps makes the transition seamless.

Virtualization

Docker is a software platform that relies on OS-level virtualization to deliver a software solution in a container. Docker exists to ease deployment of complex systems and allow environments to be evolved separately from each other. Each container is isolated from other containers, and every container can run different versions of key system libraries. Containers are small, usually needing just a few hundred MBytes of system resources. Running 10 containers or more on a single machine is common.

Docker allows you to set up an environment on one machine and move that exact environment to another machine AND run it alongside another container that has a completely different environment. For example, you might have your Product A factory test running in Java version X, and your Product B running in Java Version Y and your Product C running in C# version Z. Think of a very lightweight Virtual Machine.

You can get the tests running at your desk, and then deploy the container to your factory and run it on the same machine that is testing another

References

“Project Jupyter,” *Wikipedia*, https://en.wikipedia.org/wiki/Project_Jupyter

J. Somers, “The Scientific Paper Is Obsolete,” *The Atlantic*, April, 2018, www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676

product. The two test environments can be very different. And yet, they run alongside each other completely independently.

Debugging

The tools available to debug M2M communications are mature and pervasive. Some of the most potent are built right into your browser. But stand-alone apps (e.g., cURL) make it easy to see and understand every aspect of the communication, with detailed logs and timing. Understanding why a M2M session failed is straightforward.

Backward Compatibility

So, the capabilities might sound great. But how would legacy systems be addressed with a new approach? The answer is they'd be pulled in and used as they are today. For example, if you absolutely had to control a legacy piece of test equipment from a modern REST-based approached to factory testing, you could pull in the DLL that permitted you to control the legacy equipment in your new app. You'd probably risk some of the cross-platform compatibility at that point. But it's a reasonable trade-off.

Conclusion

Testing is, at its core, machine-to-machine communications. And some serious innovation has occurred in this area over the last two decades, fueled by the same standards and protocols that power the Internet.

Is it time to throw away the legacy systems and start over from scratch? Probably not. That ship is large and will need lots of time to turn. The investments in test systems are enormous and time is always tight when shipping a product.

But the gains to be had from state-of-the-art software (languages, analytics, and cloud storage)—much of it open source—are very real. Rather than thinking about test and measurement as a unique problem requiring unique skills and tools, think of it as the same distributed machine-to- machine communication that runs the Internet.

From this new vantage point, your test team may very well venture in a new direction for their next-gen testing solutions. And fast, low-cost, easily deployed containers that perfectly replicate your engineer's desktop environment are the outcome. And you can deploy several of these containers into your factory onto low-cost test bays running Linux.