5 From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices[1]

Dominique Guinard[1,2], Vlad Trifa[1,2], Friedemann Mattern[1], Erik Wilde[3]

[1]Institute for Pervasive Computing, ETH Zurich

[2]SAP Research, Zurich

[3]School of Information, UC Berkeley

Abstract Creating networks of "smart things" found in the physical world (e.g., with RFID, wireless sensor and actuator networks, embedded devices) on a large scale has become the goal of a variety of recent research activities. Rather than exposing real-world data and functionality through vertical system designs, we propose to make them an integral part of the Web. As a result, smart things become easier to build upon. In such an architecture, popular Web technologies (e.g., HTML, JavaScript, Ajax, PHP, Ruby) can be used to build applications involving smart things, and users can leverage well-known Web mechanisms (e.g., browsing, searching, bookmarking, caching, linking) to interact with and share these devices. In this chapter, we describe the *Web of Things (WoT)* architecture and best practices based on the RESTful principles that have already contributed to the popular success, scalability, and evolvability of the Web. We discuss several prototypes using these principles, which connect environmental sensor nodes, energy monitoring systems, and RFID-tagged objects to the Web. We also show how Web-enabled smart things can be used in lightweight ad-hoc applications, called "physical Mashups", and discuss some of the remaining challenges towards the global World Wide Web of Things.

5.1 From the Internet of Things to the Web of Things

As more and more devices are getting connected to the Internet, the next logical step is to use the World Wide Web and its associated technologies as a platform for smart things (i.e., sensor and actuator networks, embedded devices, electronic appliances and digitally enhanced everyday objects). Several years ago, in the

[1] The original publication is available at www.springerlink.com published in the book: "Architecting the Internet of Things", edited by M. Harrison, F. Michahelles and D. Uckelmann.

Cool Town project, Kindberg et al. (Kindberg et al. 2002) proposed to link physical objects with Web pages containing information and associated services. Using infrared interfaces or bar codes on objects, users could retrieve the URI of the associated page simply by interacting with the object. Another way to use the Web for real-world objects is to incorporate smart things into a standardised Web service architecture (using standards, such as SOAP, WSDL, UDDI) (Guinard et al. 2010d). In practice, this would often be too heavy and complex for simple objects.

Instead of these heavyweight Web services (SOAP/WSDL, etc.), often referred to as WS-* technologies, recent "Web of Things" projects (Wilde 2007; Guinard et al. 2010c; Luckenbach et al. 2005; Stirbu 2008) have explored simple embedded Hypertext Transfer Protocol (HTTP) servers and Web 2.0 technology. In fact, recent embedded Web servers with advanced features (such as concurrent connections or server push for event notifications), can be implemented with only 8 KB of memory and no operating system support, thanks to efficient cross-layer TCP/HTTP optimisations, and can therefore run on tiny embedded systems, such as smart cards (Duquennoy et al. 2009). Since embedded Web servers in an Internet of Things generally have fewer resources than Web clients, such as browsers or mobile phones, Asynchronous JavaScript and XML (Ajax) has proven to be a good way of transferring some of the server workload to the client.

So far, projects and initiatives, subsumed here under the umbrella term "Internet of Things", have focused mainly on establishing connectivity in a variety of challenging and constrained networking environments. A promising next step is to build scalable interaction models on top of this basic network connectivity and thus focus on the application layer. In the Web of Things concept, smart things and their services are fully integrated in the Web by reusing and adapting technologies and patterns commonly used for traditional Web content. More precisely, tiny Web servers are embedded into smart things and the REST architectural style (Richardson and Ruby 2007; Fielding 2000) is applied to resources in the physical world (Guinard et al. 2010c; Luckenbach et al. 2005; Duquennoy et al. 2009; Hui and Culler 2008). The essence of REST is to focus on creating loosely coupled services on the Web, so that they can be easily reused. REST is the architectural style of the Web (implemented by URIs, HTTP, and standardised media types, such as HTML and Extensible Markup Language (XML) and uses URIs for identifying resources on the Web. It abstracts services in a uniform interface (HTTP's methods) from their application-specific semantics and provides mechanisms for clients to select the best possible representations for interactions. This makes it an ideal candidate to build a "universal" architecture and Application Programming Interface (API) for smart things. As we will explain in this chapter, the services that smart things expose on the Web usually take the form of a structured XML document or a JavaScript Object Notation (JSON) object, which are directly machine-readable. These formats can be understood not only by machines, but are also reasonably accessible to people; provided meaningful markup elements and variable names are used and documentation is made available. They can also be

supplemented with semantic information using microformats, so that smart things can not only communicate on the Web, but also provide a user-friendly representation of themselves. This makes it possible to interact with them via Web browsers and thus explore the world of smart things with its many relationships (via links to other related things). Dynamically generated real-world data on smart objects can be displayed on such "representative" Web pages, and then processed with Web 2.0 tools. For example, things can be indexed like Web pages via their representations, users can "google" for them, and their URI can be emailed to friends or it can be bookmarked. The physical objects themselves can become active and publish blogs or inform each other using services, such as Twitter.[2] The general idea is that the Web is being used as a decentralised information system for easily exposing new services and applications, made possible, directly or indirectly, by smart things.

The Web-enablement of smart things delivers more flexibility and customisation possibilities for end-users. As an example, tech-savvy end-users, at ease with new technologies, can easily build small applications on top of their appliances. Following the trend of Web 2.0 participatory services, in particular Web Mashups (Zang et al. 2008), users can create applications mixing real-world devices, such as home appliances, with virtual services on the Web. This type of applications is often referred to as physical Mashup (Wilde 2007, Guinard et al. 2010c). As an example, a music system could be connected to Facebook or Twitter in order to post the songs one mostly listens to. On the Web, this type of small, ad-hoc application is usually created through a Mashup editor (e.g., Yahoo Pipes[3]), which is a Web platform that enables tech-saavy users (i.e., proficient users of technology) to visually create simple rules to compose Web sites and data sources. We describe how these principles and tools can also be applied to empower the user to create physical Mashups on top of their things.

In Section 2 and 3 we provide a "cookbook" describing the design steps towards embedding smart things into the Web. We also discuss a number of patterns and illustrate them via real prototypes that we have developed over the past few years. In Section 4, we use three concrete prototypes to exemplify how developers, domain-experts, and tech-savvy users can all benefit from a composable Web of Things. Finally, in Section 5 and 6 we discuss the remaining challenges towards implementing a World Wide Web of Things.

[2] http://www.twitter.com

[3] http://pipes.yahoo.com/pipes/

5.2 Designing RESTful Smart Things

The "Web of Things" can be realised by applying principles of Web architecture, so that real-world objects and embedded devices can blend seamlessly into the Web. Instead of using the Web as a transport infrastructure – as done when using WS-* Web services – we aim at making devices an integral part of the Web and its infrastructure and tools by using HTTP as an application layer protocol.

The main contribution of the "Web of Things" approach is to offer a foundation for the next step beyond basic network connectivity. We hope that the Web of Things can do for real-world resources what the Web did for information resources: basic connectivity was a necessary, but not a sufficient condition for the Internet to grow as spectacularly as it is still growing today; it was the architecture of the Web that allowed data and services to be shared in a way that was unheard of before, and that spurred the decentralised growth of what was made available on the Web.

In this section, we describe the use of REST (Fielding 2000) as a universal interaction architecture, so that interactions with smart things can be built around universally supported methods (Pautasso and Wilde 2009).

In the following, we provide a set of guidelines to Web-enable smart things and illustrate them with concrete examples of implemented prototypes. As case study, we describe how we Web-enabled a wireless sensor network (Sun SPOT[4]). These guidelines are based on the concepts of *Resource Oriented Architecture (ROA)*, described by Richardson and Ruby (Richardson and Ruby 2007). Our main goal is to focus on how these concepts can be applied and adapted in order to apply to smart things.

5.2.1 Modeling Functionality as Linked Resources

The central idea of REST revolves around the notion of a resource as *any component of an application that is worth being uniquely identified and linked to*. On the Web, the identification of resources relies on Uniform Resource Identifiers (URIs), and representations retrieved through resource interactions contain links to other resources, so that applications can follow links through an interconnected web of resources. Clients of RESTful services are supposed to follow these links, just like one browses Web pages, in order to find resources to interact with. This allows clients to "explore" a service simply by browsing it, and in many cases, services will use a variety of link types to establish different relationships between resources.

[4] http://www.sunspotworld.com

In the case of the Sun SPOT, each node has a few sensors (light, temperature, accelerometer, etc.), actuators (digital outputs, LEDs, etc.), and a number of internal components (radio, battery). Each of these components is modeled as a resource and assigned a URI. For instance, typing a URI such as

```
http://.../sunspots/spot1/sensors/light
```

in a browser requests a representation of the resource *light* of the resource *sensors* of *spot1*. Resources are primarily structured hierarchically and each resource also provides links back to its parent and forward to its children. As an example, the resource

```
http://.../sunspots/spot1/sensors/
```

provides a list of links to all the sensors offered by *spot1*. This interlinking of resources that is established through both, resource links and hierarchical URI, is not strictly necessary, but well-designed URIs make it easier for developers to "understand" resource relationship and even allow non-link based "ad-hoc interactions", such as "hacking" a URI by removing some structure and still expecting for it to work somehow.[5]

In a nutshell, the first step when Web-enabling a smart thing is to design its *resource network*. Identification of resources and their relationships are the two important aspects of this step.

5.2.2 Representing Resources

Resources are abstract entities and are not bound to any particular representation. Thus, several formats can be used to represent a single resource. However, agreed-upon resource representation formats make it much easier for a decentralised system of clients and servers to interact without the need for individual negotiations. On the Web, media type support in HTTP and the Hypertext Markup Language (HTML) allow peers to cooperate without individual agreements. It further allows clients to navigate amongst the resources using hyperlinks.

For machine-to-machine communication, other media types, such as the XML and the JSON have gained widespread support across services and client plat-

[5] In some browsers this "URI hacking" is even part of the UI, where a "go up" function in the browser simply removes anything behind the last slash character in the current URI and expects that the Web site will serve a useful representation at that guessed URI.

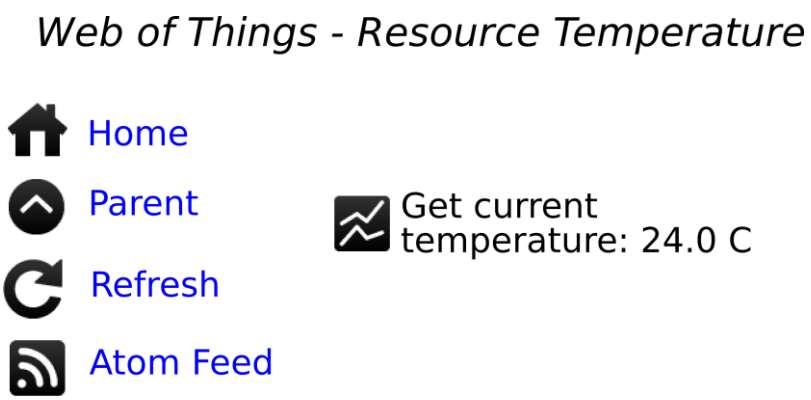
forms. JSON is a lightweight alternative to XML that is widely used in Web 2.0 applications.[6]

In the case of smart things, we suggest support for at least an HTML representation to ensure browsability by humans. Note that since HTML is a rather verbose format, it might not be directly served by the things themselves, but by intermediate proxies, as described in Section 0. For machine-to-machine communications, we suggest using JSON. Since JSON is a more lightweight format compared to XML, we believe that it is better adapted to devices with limited capabilities such as smart things. Furthermore, it can directly be parsed to JavaScript objects. This makes it an ideal candidate for integration into Web Mashups.

In the Sun SPOT example, each resource provides both, an HTML and a JSON representation. As an example, the listing in Figure 5.1a shows the JSON representation of the *temperature* resource of a Sun SPOT and Figure 5.1b shows the same resource represented as an HTML page with links to parents, subresources, and related resources.

```
1     {"resource":
2     {"methods":["GET"],
3     "name":"Temperature",
4     "children":[],
5     "content":
6     [{"description":"Current Temperature",
7     "name":"Current Ambient Temperature",
8     "value":"27.75"}]}}
```

Fig. 5.1a JSON Representation of the Temperature Resource of a Sun SPOT



[6] http://www.json.org

Fig. 5.1b HTML Representation (Rendered by a Browser) of the Temperature Resource of a Sun SPOT Containing Links to Parent and Related Resources

5.2.3 Servicing Through a Uniform Interface

In REST, interacting with resources and retrieving their representations all happens through a uniform interface which specifies a service contract between the clients and servers. The uniform interface is based on the identification (and thus interaction) of resources, and in case of the Web, this interface is defined by the HTTP. We concentrate on three particular parts of this interface: operations, content-negotiation, and status codes.

5.2.3.1 Operations

HTTP provides four main methods to interact with resources, often also referred to as "verbs": *GET*, *PUT*, *POST*, and *DELETE*. *GET* is used to retrieve the representation of a resource. *PUT* is used to update the state of an existing resource or to create a resource by providing its identifier. *POST* creates a new resource without specifying any identifier. *DELETE* is used to remove (or "unbind") a resource.

In the Web of Things, these operations map rather naturally, since smart things usually offer quite simple and atomic operations. As an example, a *GET* on

```
http://.../spot1/sensors/temperature
```

returns the temperature observed by *spot1*, i.e., it retrieves the current representation of the temperature resource. A *PUT* on

```
http://.../sunspots/spot1/actuators/leds/1
```

with the updated JSON representation {"*status":"on"}* (which was first retrieved with a GET on */leds/1*) switches on the first LED of the Sun SPOT, i.e., it updates the state of the LED resource. A *POST* on

```
http://.../spot1/sensors/temperature/rules
```

with a JSON representation of the rule as {"*threshold":35}* encapsulated in the HTTP body, creates a rule that will notify the caller whenever the temperature is higher than 35 degrees, i.e., it creates a new rule resource without explicitly providing an identifier. Finally, a *DELETE* on

```
http://.../spot
```

is used to shutdown the node, or a *DELETE* on

```
http://.../spot1/sensors/temperature/rules/1
```

is used to remove rule number 1.

Additionally, another less-known verb is specified in HTTP and implemented by most Web servers: *OPTIONS* can be used to retrieve the operations that are allowed on a resource. In a programmable Web of Things, this feature is quite useful, since it allows applications to find out at runtime what operations are allowed for any URI. As an example, an *OPTIONS* request on

```
http://.../sunspots/spot1/sensors/tilt
```

returns *GET, OPTIONS*.

5.2.3.2 Content Negotiation

HTTP also specifies a mechanism for clients and servers to communicate about the requested and provided representations for any given resource; this mechanism is called *content negotiation*. Since content negotiation is built into the uniform interface of HTTP, clients and servers have agreed-upon ways in which they can exchange information about requested and available resource representations, and the negotiation allows clients and servers to choose the best representation for a given scenario.

A typical content-negotiation for the Sun SPOTs looks as follows. The client begins with a *GET* request on

```
http://.../spot1/sensors/temperature/rules
```

It also sets the *Accept* header of the HTTP request to a weighted list of media types it understands, for example to: *application/json;q=1*, *application/xml;q=0.5*. The server then tries to serve the best possible format it knows about and specifies it in the *Content-Type* of the HTTP response. In our case, the Sun SPOT cannot offer XML and would thus return a JSON representation and set the HTTP header *Content-Type: application/json*.

5.2.3.3 Status Codes

Finally, the status of a response is represented by standardised status codes sent back as part of the header in the HTTP message. There exist several dozens of codes which each have well-known meaning for HTTP clients. In a Web of

Things, this is very valuable since it gives us a lightweight but yet powerful way of notifying abnormal requests execution.

As an example, a *POST* request on

```
http://.../sunspots/spot1/sensors/acceleration
```

returns a 405 status code that the client has to interpret as the notification that "the method specified in the request is not allowed for the resource identified by the request URI."

5.2.4 Syndicating Things

Many applications for smart things require syndicating information about objects or collections of objects. With Atom, the Web has a standardised and RESTful model for interacting with collections, and the Atom Publishing Protocol (AtomPub) extends Atom's read-only interactions with methods for write access to collections. Because Atom is RESTful, interactions with Atom feeds can be based on simple GET operations which can then be cached. Atom enables decoupled scenarios by allowing clients to monitor smart things by subscribing to feeds and polling a feed on a remote server, instead of directly polling data from each device.

We implemented this model for the Sun SPOTs, since it fits the interaction model of sensor networks. Thus, the nodes can be controlled (e.g., turning LEDs on, enabling the digital outputs, etc.) using synchronous HTTP calls (client pull) as explained before, but can also be monitored by subscribing to feeds (node push). For example, a subscription to a feed can be done by creating a new "rule" on a sensor resource and *POST*ing a threshold (e.g., > 100).

```
http://.../sunspots/spot1/sensors/light/rules
```

In response, the Sun SPOT returns a URI to an Atom feed. Every time the threshold is reached, the node pushes a JSON message to the Atom server using AtomPub. This allows for thousands of clients to monitor a single sensor by *outsourcing* the processing onto an intermediate, more powerful server.

5.2.5 Things Calling Back: Web Hooks

While Atom allows asynchronous communication between clients and smart things, clients still need to pull the feed server on a regular basis to get data. In addition to being inefficient in terms of communications, this might be problematic

for scenarios where the focus is on monitoring. This is often the case with applications communicating with wireless sensor networks.

For those applications, we suggest supporting HTTP callbacks, sometimes called *Web hooks*.[7] *Web hooks* are a mechanism for clients and applications that want to receive notifications from other Web sites using user-defined callbacks over HTTP. Users can specify a callback URI where the application will POST data to once an event occurs. This mechanism has been used by the PayPal service which allows you to specify a URI to be triggered by the service once payment has been accepted.

As an example, let us consider again the case of creating a new rule on a Sun SPOT:

```
http://.../sunspots/spot1/sensors/light/rules
```

Now, alongside with the rule, the client *POST*s a URI on which it will listen for incoming messages. Every time the threshold is reached, the node (or an intermediate) will push a JSON message to the given URI(s).

Using Web hooks is a first step towards bi-directional, real-time interaction with smart things. However, this model has a number of limitations as it requires from clients to have a public URI where data can be posted to, which is rarely the case when clients are behind a firewall. We will discuss further solutions in Section 0.

5.3 Web-enabling Constrained Devices

Although Web servers are likely to be embedded into more and more devices, we cannot assume that every smart device will directly offer a RESTful interface. In some cases, it makes sense to hide the platform-dependent protocol to access the resources of a particular device, and to expose them as RESTful service provided by a gateway. The actual interactions behind that RESTful service are invisible and often will include specialised protocols for the specific implementation scenario. REST defines the notion of *intermediaries* as a core part of the architectural style, and therefore such a design can easily be achieved by implementing the RESTful service on intermediaries. By using either *proxies* or *reverse proxies*, it is furthermore possible to establish such an intermediary from the client or from the server side, effectively introducing a robust pattern for wrapping non-RESTful services in RESTful abstractions.

In practice, two solutions are possible: Web connectivity directly on the smart things, or indirectly through a proxy. Previous work has shown that serving con-

[7] http://www.webhooks.org

tent using Web servers on resource-constrained devices is feasible (Duquennoy et al. 2009). Also, in the foreseeable future, most embedded platforms will have native support for TCP/IP connectivity (in particular with 6LowPAN (Hui and Culler 2008), therefore, a Web server on most devices is a reasonable assumption. This approach is sometimes desirable, as there is no need to translate HTTP requests from Web clients into the appropriate protocol for the different devices, and thus devices can be directly integrated and make their RESTful APIs directly accessible on the Web, as shown in the right part of Figure 5.2.

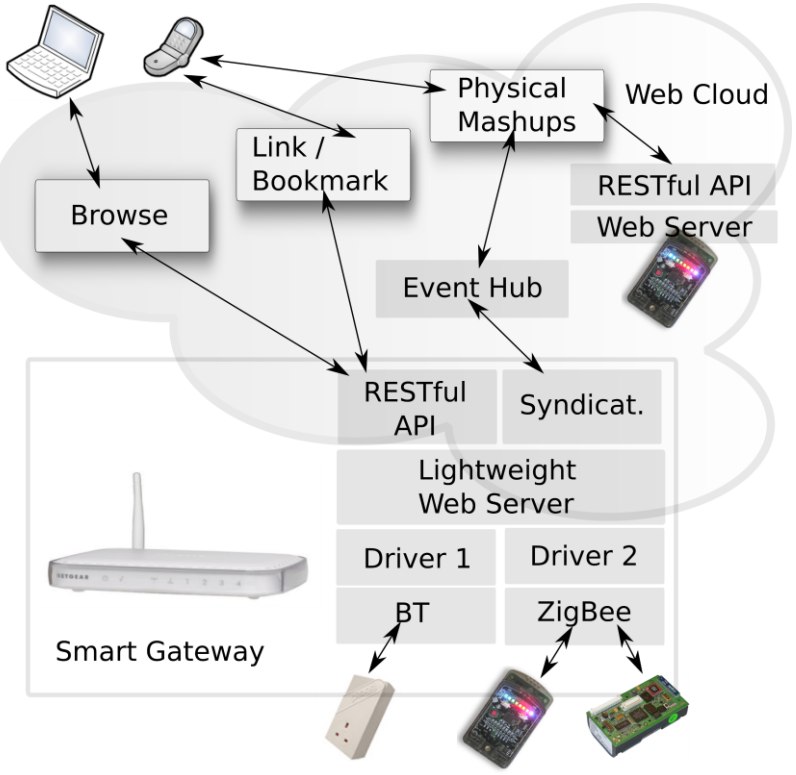


Fig. 5.2 Web and Internet Integration with Smart Gateways and Direct Integration

However, when an on-board HTTP server is not possible or not desirable, Web integration takes place using a reverse proxy that bridges devices that are not directly accessible as Web resources. We call such as proxy a *Smart Gateway* (Trifa et al. 2009) to account for the fact that it is a network component that does more than only data forwarding. A Smart Gateway is a Web server that hides the actual communication between networked devices (e.g., Bluetooth or Zigbee) and the clients through the use of dedicated drivers behind a RESTful service. From the

Web clients' perspective, the actual Web-enabling process is fully transparent, as interactions are HTTP in both cases.

As an example, consider a request to a sensor node coming from the Web through the RESTful service. The gateway maps this request to a request into the proprietary API of the node and transmits it using the communication protocol understood by the sensor node. A Smart Gateway can support several types of devices through a driver architecture, as shown in Figure 5.3, where the gateway supports three types of devices and their corresponding communication protocols. Ideally, gateways should have a small memory footprint to be integrated into embedded computers already present in network infrastructures, such as wireless routers, set-top boxes, or Network Attached Storage (NAS) devices.

Aside from connecting limited devices to the Web, a Smart Gateway can also provide more complex functions to devices such as orchestration and composition of several low-level services, offered by various devices into higher-level services available through the RESTful service. For example, if an embedded device measures the energy consumption of appliances, the Smart Gateway could provide a service that returns the total energy consumption as a sum of the data collected by all the devices connected to the gateway. Additionally, a gateway could take care of notifying all the URI call-backs (or Web hooks) whenever a given condition is met.

Example: A Smart Gateway for Smart Meters
A prototype for a smart meter infrastructure illustrates the application of the WoT architecture and the concept of Smart Gateways for monitoring and controlling the energy consumption of households. We used intelligent power sockets, called *Plogg*[8], which can measure the electricity consumption of the appliance plugged into them. Each Plogg is also a wireless sensor node that communicates over Bluetooth or Zigbee. However, the integration interface offered by the Ploggs is proprietary, which makes the development of applications using Ploggs rather tedious, and does not allow for easy Web integration.

[8] http://www.plogginternational.com

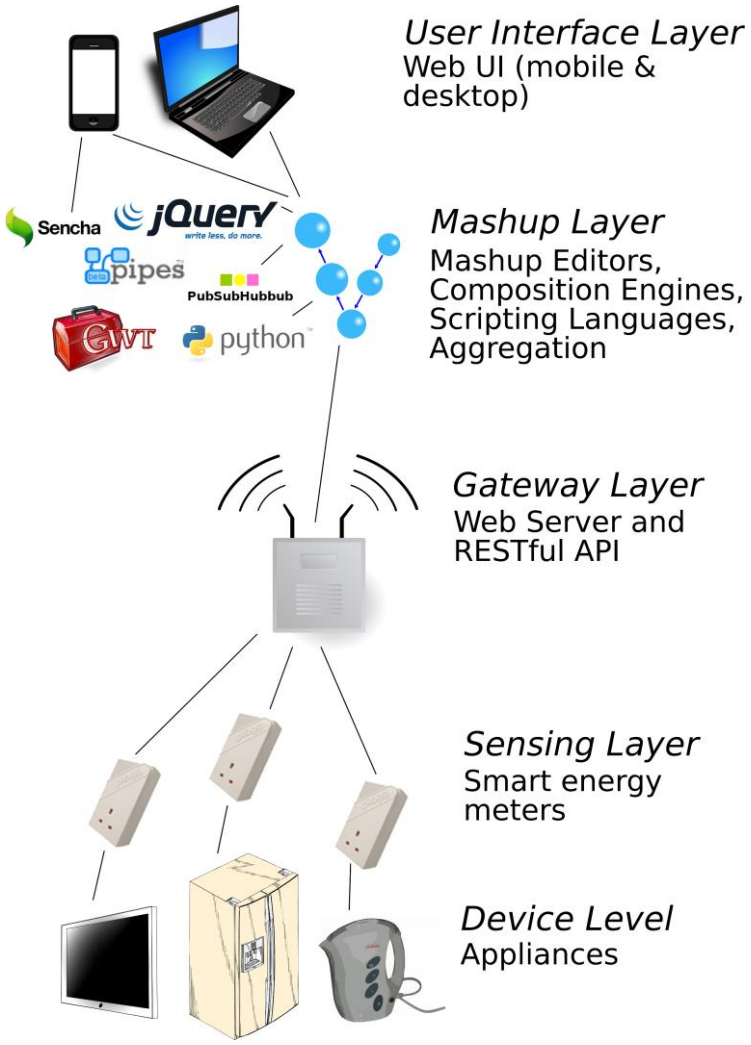


Fig. 5.3 Appliances Attached to Ploggs Power Outlets Which Communicate with a Smart Gateway Offering the Ploggs' Functionalities as RESTful Web Services

The Web-oriented architecture we have implemented using the Ploggs is based on five main layers as shown in Figure 5.3. The Device Layer is composed of appliances we want to monitor and control through the system. In the Sensing Layer, each of these appliances is then plugged into a Plogg sensor node. In the Gateway Layer, the Ploggs are discovered and managed by a Smart Gateway as described before. In the Mashup layer the Ploggs' services are composed together to create an energy monitoring and control application, using Web scripting languages or composition tools. Finally, this application is made available through a Web User

Interface in a Web browser (e.g., on a mobile phone, a desktop computer, a tablet PC, etc.)

The Smart Gateway in this example is a C++ application running on an embedded machine, whose role is to automatically find all the Ploggs in the environment and make them available as Web resources. The gateway first periodically looks for the Ploggs in the area by scanning the environment for Bluetooth devices. The next step is to expose them as RESTful resources. A small footprint Web server (Mongoose[9]) is used to enable access to the Ploggs' functionalities over the Web, simply by mapping URIs to the various requests of the native Plogg Bluetooth API.

In addition to discovering the Ploggs and mapping their functionalities to URIs, the Smart Gateway has two other important features. First, it offers *local* aggregates of device-level services. For example, the gateway offers a service that returns the combined electricity consumption of all the Ploggs found at any given time. The second feature is that the gateway can represent resources in various formats. By default an HTML page with links to the resources, is returned, this ensures browsability. Using this representation the user can literally "browse" with any Web client the structure of smart meters to identify the one he or she wants to use and directly test the Ploggs by clicking on links (e.g., for the HTTP *GET* method) or filling forms (e.g., for the *POST* method). Alternatively, the Smart Gateway can also represent results of resources like JSON, to ease the integration with other Web applications.

To illustrate the concept from a client point of view, let us briefly describe an example of interaction between a client application (e.g., written in Ajax) and the Ploggs' RESTful Smart Gateway. First, the client contacts the root URI of the application

```
http://.../EnergieVisible/SmartMeters/
```

with the *GET* method. The server responds with the list of all the smart meters connected to the gateway.

Afterwards, the client selects from that list the device it wants to interact with identified by a URI

```
http://.../EnergieVisible/SmartMeters/RoomLamp
```

alongside with the format it wants to get back (using HTTP content negotiation, see Section 5.2.3). By issuing a *GET* request on this resource with the *Accept* header set to *application/json;q=1,* it gets back a JSON representation as shown in Figure 5.4 below. In the response message of this listing, the client finds energy

[9] http://code.google.com/p/mongoose

consumption data (e.g., current consumption, global consumption, etc.) as well as hyperlinks to related resources. Using these links, the client can discover other related "services".

```
1     GET /EnergieVisible/SmartMeters/RoomLamp
2     [...] HTTP/1.x 200 OK
3     Content-Type: application/json
4     {
5       "deviceName":  "RoomLamp",
6       "currentWatts":  60.52,
7       "KWh":  40.3,
8       "maxWattage":  80.56
9       "links":
10      [{"aggregate":  "../all"},
11      {"load": "../load"},
12      {"status":  "/status"}]
13    }, {...}]
```

Fig. 5.4 JSON Representation of a Plogg connected to a Lamp

As an example, by contacting

```
http://.../RoomLamp/status
```

with the standard OPTIONS method, the client gets back a list of the methods allowed on the status resource (e.g., *Allow: GET, HEAD, POST, PUT*). By sending the *PUT* method to this URI alongside with the representation (e.g., JSON) {"*statu":"off"}*, the appliance plugged into the Plogg is turned off.

The Web-enabling of the Ploggs through a Smart Gateway allows building fully Web-based energy monitoring applications. It also enables simple interactions, such as bookmarking connected appliances, and control or monitor them from any device (e.g., a mobile phone, an embedded computer, a wireless sensor node, etc.), offering a standard Web browser or understanding the HTTP protocol.

5.4 Physical Mashups: Recomposing the Physical World

In this section, we illustrate how the Web of Things concepts and architecture facilitates the creation of Mashups in the physical world. A Web Mashup is an application that takes several Web resources and uses them to create a new application. Unlike traditional forms of integration, Mashups focus mainly on opportunistic integration occurring on the Web for an end-user's personal use and generally for non-critical applications (Yu et al. 2008). They are usually created

ad-hoc, using lightweight and well-known Web technologies, such as JavaScript and HTML, and contribute to serving short terms needs. As an example, a Mashup can be created to display, on Google Maps, the location of all the pictures posted to Flickr.[10]

By extending the Mashup concept to physical objects and applying RESTful patterns to smart things, we allow their seamless integration into the Web, thus enabling a new range of applications based on this unified view of a Web of information resources and physical objects. We call this concept "physical Mashup", because it is directly inspired from Web 2.0 Mashups.

In this section, we present three Mashups representing three different use cases. In the first prototype, we create an energy monitoring and control system based on the Ploggs Smart Gateway. In the second, we show how domain experts (e.g., product managers, marketing executives, etc.) can leverage such tools to build a business intelligence platform suited to their business needs. In the last example, we show how end-users could use a visual physical Mashup editor to dynamically "configure" their home appliances.

[10] http://www.flickr.com

5.4.1 Energy Aware Mashup: "Energie Visible"

Fig. 5.5 The Web-Based User Interface for Monitoring and Controlling the Ploggs

In this first example, we create a Mashup to help households to understand their energy consumption and to be able to remotely monitor and control it.

The idea of the "Energie Visible"[11] project is to offer a Web dashboard that enables people to visualise and control the energy consumption of their household appliances. The dashboard is shown in Figure 5.5 and provides six real-time interactive graphs. The four graphs on the right side provide detailed information about the current electricity consumption of all the detected Ploggs.

Thanks to the Ploggs Smart Gateway described before, the dashboard can be implemented using any Web scripting language or tool (PHP, Ruby, Python, JavaScript, etc.). The Energie visible application was built using Google Web Toolkit (GWT)[12], which is a platform for developing JavaScript Web applications in Java, and provides a large number of easily customisable widgets. To display the current energy consumption in real time, the application simply sends HTTP *GET* requests to the gateway

[11] The project is available on http://www.webofthings.com/energievisible

[12] http://code.google.com/webtoolkit/

```
http://.../EnergieVisible/SmartMeters/all.json
```

on a regular basis or subscribes to this resource using Web hooks. The resulting feed entry is then dispatched to the corresponding graphs widgets, which can directly parse JSON, and extract the relevant data in it to be displayed.

The "Energie Visible" prototype was deployed at the headquarters of a private foundation working on sustainability (cudrefin02[13]) and has now been running reliably since November 2008.

The aim of the project was to help visitors and members to better understand how much each device consumes in operation and in standby. The Ploggs are used to monitor the energy consumption of various devices, such as a fridge, a kettle, several printers, a file server, computers and screens. A large display in the office enables people passing by to experiment with the energy consumption of the devices. The staff can also access the user interface of any Plogg with the Web browser of their office computer.

5.4.2 Business Intelligence Mashup: RESTful EPCIS

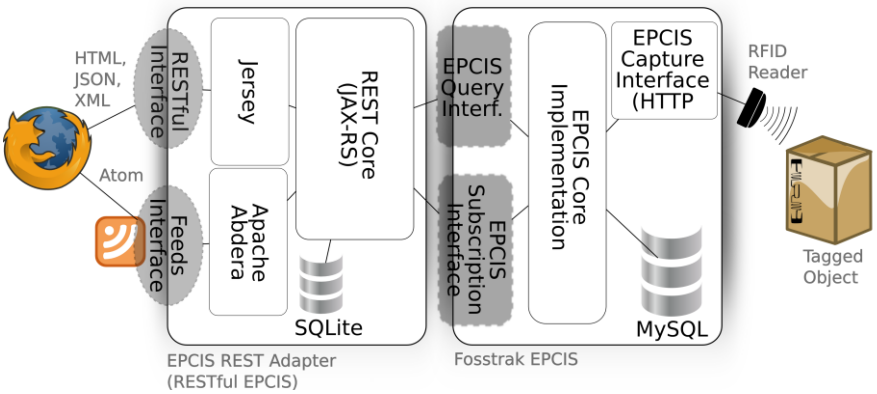


Fig. 5.6 Architecture of the RESTful EPCIS Based on the Jersey RESTful Framework and Deployed on Top of the Fosstrak EPCIS

The Electronic Product Code (EPC) Network (Floerkemeier et al. 2007) is a set of standards established by industrial key players towards a uniform platform for tracking and discovering RFID-tagged objects and goods in supply chains. This network offers a standardised server-side EPC Information Service (EPCIS) for

[13] http://cudrefin02.ch

managing and offering access to track and trace RFID events. Implementations of EPCIS provide a standard query and capture API through WS-* Web Services.

In order to integrate not only embedded devices, but also RFID-tagged everyday items into the Web of Things, we use the concepts presented to turn the EPCIS into a "Smart Gateway". This helps to better grasp the benefits of a seamless Web integration based on REST, as opposed to using HTTP as a transport protocol only (as WS-* Web Services use it).

The EPCIS offers three core features. First, it offers an interface to query for RFID events. The WS-* interface, however, does not allow to directly query for RFID events using Web languages, such as JavaScript or HTML. More importantly, it does not allow to explore the EPCIS using a Web browser, or to search for tagged objects or exchange links pointing to traces of tagged objects. To remedy the problem, we implemented a RESTful translation of the EPCIS WS-* interface.

As shown in Figure 5.6, the RESTful EPCIS (Guinard et al. 2010b) is a software module based on Jersey[14], a software framework for building RESTful applications. Clients of the RESTful EPCIS, such as browsers or Web applications, can query for tagged objects directly using REST and its uniform HTTP interface. Requests are then translated by the framework into WS-* calls on the standard EPCIS interface. This allows for the RESTful EPCIS to serve data provided by any implementation of the EPCIS standard. In our case we use Fosstrak (Floerkemeier et al. 2007)[15], an open source implementation of the standard.

The first benefit of the RESTful EPCIS is that every RFID event, reader, tagged object or location is turned into a Web resource and gets a globally resolvable URI, which uniquely identifies it and can be used to retrieve various representations. EPCIS queries are transformed into compositions of these identifiers and can be directly executed in the browser, sent by email, or bookmarked. As an example, a factory manager who wants to know what tagged objects enter his factory can bookmark a URI, such as:

```
http://.../epcis/rest/location/urn:company:factory1/reader/urn:company:entra
nce:1
```

Furthermore, these URIs are linked together through their representations in order to reflect the relationships of the physical world. This makes the RESTful EPCIS directly browsable. Indeed, in addition to the XML representation of tagged objects offered by the standard, it also provides HTML, JSON and Atom representations. With the HTML representation, end-users can literally browse tagged things and their traces simply by following hyperlinks in the very same

[14] http://jersey.dev.java.net

[15] http://www.fosstrak.org

way as they browse the Web of documents. For example, a location offers links to co-located RFID readers.

With the Atom representation, end-users can formulate queries by browsing the hyperlinked EPCIS and obtain the updated results represented as Atom feeds, which browsers can understand and directly subscribe, too. As an example, a product manager can create a feed in order to be automatically updated in his browser whenever one of his products is ready to be shipped. He can then use the URI of the feed to send it to his most important customers so that they could track the goods' progress as well. This is a simple but very useful use case, which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

5.4.3 A Mashup Editor for the Smart Home

Tech-savvy users can create Web Mashups using "Mashup editors", such as Yahoo Pipes. These editors usually provide visual components representing Web sites and operations (add, filter, etc.) that users only needs to connect (or *pipe*) together to create new applications. We wanted to apply the same principles to allow users to create physical Mashups without requiring any programming skills.

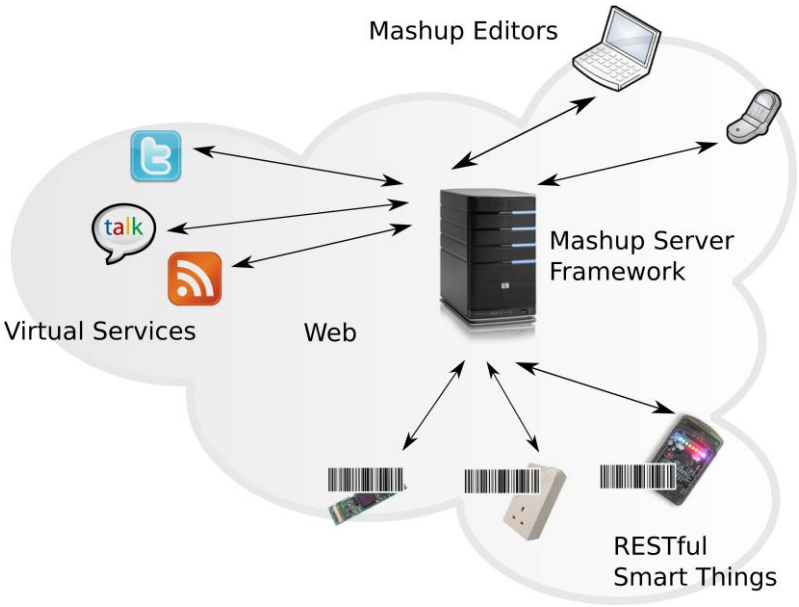


Fig. 5.7 The Physical Mashup Framework

We briefly introduce our physical Mashup architecture and two Mashup editors built on top of it. As shown in Figure 5.7, the system is composed of four main parts. We first have RESTful, Web-enabled, smart things and appliances. In our prototype, we tag them with small 2D barcodes in order to ease their identification with mobile phones. We then have "virtual" services on the Web, such as Twitter, Google Visualisation API, Google Talk, etc. In the middle, the Mashup server framework allows to compose services of different smart appliances as well as virtual services on the Web. It is in charge of executing the workflows created by end-users in their Mashup applications. It discovers, listens, and interacts with the devices over their RESTful API. The last components are the Mashup editors themselves, which allow users to create Mashup applications very easily.

We implemented two Mashup editors using this architecture. The first one is based on the Clickscript project.[16] A Firefox plugin written on top of an Ajax library allows people to visually create Web Mashups by connecting building blocks of resources (Web sites) and operations (greater than, if/then, loops, etc.). Since it is written in JavaScript, Clickscript cannot use resources based on proprietary service protocols. However, it can easily access RESTful services, such as those provided by Web-enabled smart appliances. This makes it straightforward to create Clickscript building blocks that represent smart appliances. The Mashup shown in Figure 5.8 gets the room temperature by *GET*ting the temperature resource. If it is below 36 degrees, it turns off the Web-enabled air-conditioning system.

The second editor was implemented on the Android Mobile Phone. Once again, thanks to the support of HTTP in Android, RESTful communication with smart appliances was straightforward. Similarly to Clickscript, the mobile editor allows the creation of simple Mashups. However, due to the screen constraints of the mobile phone, a Mashup is created by going through a wizard. Users first select the appliances they want to include in the Mashup. They do this simply by scanning a barcode on the appliance using the phone's camera. These codes are basically pointing back to the root URLs of the appliance's RESTful APIs. They then set up the rules they want to implement and the virtual services they want to interact with. For example, users can create a Mashup that switches on their appliances, e.g, turning the heating up, whenever their phone detects that they are moving towards home (based on their GPS traces).

[16] http://www.clickscript.ch

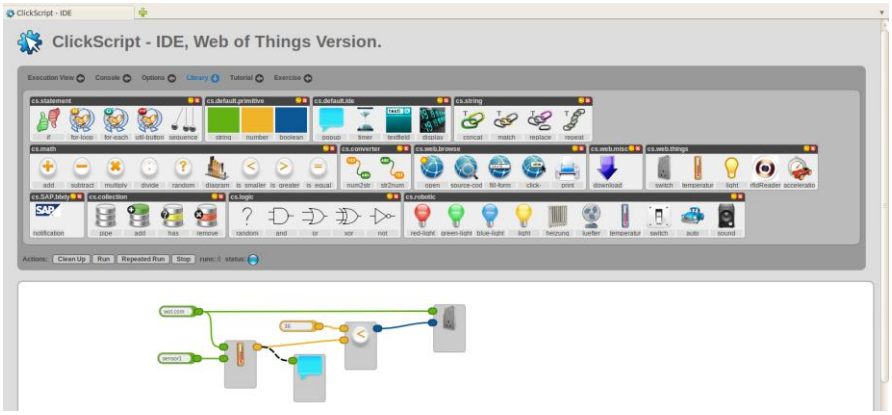


Fig. 5.8 Using the Clickscript Mashup Editor to Create a Physical Mashup by Connecting Building Blocks Directly to a Browser

5.5 Advanced Concepts: The Future Web of Things

So far, we have shown how Web standards and design principles can be leveraged for smart things. While this seems to be a rather adequate architecture for the Web of Things, many open challenges remain. In this section, we explore three such challenges, and sketch potential solutions for each. We begin by discussing the needs for *real-time* data of many smart things applications. Then, we address the challenges of *finding* and *understanding* services available in a global Web of Things. We finally look at mechanisms for *sharing* smart things.

5.5.1 Real-time Web of Things

HTTP is a stateless client/server protocol where interactions are always initiated by the client, and there is no protocol context bigger than a request/response exchange. This interaction model is well-suited for control-oriented applications where clients read/write data from/to embedded devices. However, this client-initiated interaction models seem inappropriate for bi-directional event-based and streaming systems, where data must be sent asynchronously to the clients as soon as it is produced.

For example, many pervasive scenarios must deal with real-time information to combine stored or streaming data from various sources to detect spatial or temporal patterns, as is the case in many environmental monitoring applications. As such applications are often event-based and embedded devices usually have a low-duty

cycle (i.e., sleep most of the time), smart things should also be able to push data to clients (rather than being continuously polled). To support the complex, data-centric queries required for such scenarios, more flexible data models are required to expose sensor data streams over the Web. In this section, we explore the recent developments in the real-time Web to build such a data model that is more suited to the data-centric, stream-based nature of sensor-driven applications.

As mentioned before, using syndication protocols, such as Atom, improves the model when monitoring, since devices can publish data asynchronously using AtomPub on an intermediate server or Smart Gateway. Nevertheless, clients still have to pull data from Atom servers. Web streaming media protocols (RTP/RTSP) have enabled transmission of potentially infinite data objects, such as Internet radio stations. Sensor streams are similar to streaming media in this respect. However, streaming media mainly support *play* and *pause* commands, which are insufficient for sensor streams where more elaborate control commands are needed. The Extensible Messaging and Presence Protocol (XMPP)[17] is an open standard for real-time communication based on exchanges of XML messages, and powers a wide range of applications including instant messaging (Google Talk is based on XMPP). Although widely used and successful, XMPP is a fairly complex standard, which is often too heavy for the limited resources of embedded devices used in sensor networks.

An alternative type of Web applications that attempt to eliminate the limitations of the traditional HTTP polling has become increasingly popular. This model, called *Comet*[18] (also called HTTP streaming or server push), enables a Web server to push data back to the browser without the client requesting it explicitly. Since browsers are not designed with server-sent events in mind, Web application developers have tried to work around several specification loopholes to implement Comet-like behavior, each with different benefits and drawbacks. One general idea is that a Web server does not terminate the TCP connection after response data has been served to a client, but leaves the connection open to send further events.

Based on this brief overview, one can observe that the tradeoff between scalability and query expressiveness is also present in the Web world. However, as the recent developments in Web techniques have allowed to build efficient and scalable publish/subscribe systems, we suggest that a Web-based pub/sub model could be used to connect sensor networks with applications. PubSubHubbub (PuSH)[19] is a simple, open pub/sub protocol as an extension to Atom and RSS. Parties (servers) speaking the PuSH protocol can get near-instant notifications (via callbacks) when

[17] http://www.xmpp.org

[18] http://www.tinyurl.com/tc95h

[19] http://code.google.com/p/pubsubhubbub

a feed they are interested in is updated. PuSH can also be used as a general-purpose messaging protocol for devices (Trifa et al. 2010).

The following model can be used to enable Web-based stream processing applications where users can post queries using an HTTP request to one or more sensors. The HTTP request shown in Figure 5.9, collects the light and temperature sensor readings twice per second (the `ds.freq=2` Hz parameter) only if the light sensor value is not over "200" and the temperature reading is less than "19":

```
1     POST /datastreams/ HTTP/1.1
2     Content-Type:
      application/x-www-form-urlencoded
3
4     ds.device=purpleSensor
5     &ds.data=temperature,light
6     &ds.freq=2
7     &ds.filter=light <= 200 && temperature < 19
```

Fig. 5.9 HTTP Request Collecting Light and Temperature Sensor Readings

As a result, a specific pub/sub feed will be created on a pub/sub broker as a stream (sequence of messages) in which all the data matching the request will be pushed by the stream processing engine. This allows decoupling the application from the stream processing engine, which can be easily replaced, as long as it supports the same interface to process Web requests and also can push the matching data into the pub/sub broker.

All the data samples corresponding to these queries are then pushed into a feed on the message broker, where users can subscribe using the PuSH protocol. They will then receive the data from the stream pushed from the broker via callbacks.

Although HTTP was not designed for real-time stream delivery, exploratory research in the Web of Things area shows promising results when using Web standards to interact with distributed sensors and actuators (Trifa et al. 2010). The loss in raw performance and latency, due to verbose HTTP requests, is compensated by allowing sensor networks to be exposed in an easily accessible and universal way. Additionally, thanks to the many advantages offered by Web standards, such as transparent proxies, declarative Web-based queries can be mapped to the specialised processing features of sensor networks, therefore, one can still take advantage of the optimisations and advanced processing implemented within sensor networks and other stream processing systems.

While it is clear that a Web of Things needs more developments and standards in the areas that we have described, the developments of recent years and the foreseeable future of HTML5 and its Web Sockets and Server-Sent Events is a sign of

developments moving in the right direction for the WoT. However, it is an important task for Internet of Things researchers to identify the shortcomings of the current Web architecture and propose solutions that work well for monitoring the real world and still integrate well with the Web.

5.5.2 Finding and Describing Smart Things

Another major challenge for a global Web of Things is searching and finding relevant devices among billions of smart things that will be connected to the Web. Finding them by browsing HTML pages with hyperlinks is literally impossible in this case, hence the idea of searching for smart things. Searching for things is significantly more complicated than searching for documents, as things are tightly bound to contextual information, such as location, are often moving from one context to another, and have no obvious easily indexable properties, such as human-readable text in the case of documents.

Beyond location, smart things need a mechanism to describe themselves and their services to be (automatically) discovered and used. But what is the best way to describe a thing on the Web so that both, humans and machines, can understand what services it provides? This problem is not inherent to smart things, but more generally a complex problem of describing services, which has always been an important challenge to be tackled in the Web research community, usually in the area of the *Semantic Web*.[20]

To overcome the rather limited descriptive power of resources on the Web, several languages have been proposed, such as RDF[21] or Microformats[22]. Designed for both, human and machines, Microformats provide a simple way to add semantics to Web resources. There is not one single Microformat, but rather a number of them, each one for a particular domain; a “geo” and “adr” microformat for describing places or an “hProduct” and “hReview” microformat for describing products and what people think about them. Each Microformat undergoes a “standardisation” process that ensures its content to be widely understood and used, if accepted.

Microformats are especially interesting in a Web of Things for two reasons; first they are directly embedded into Web pages and thus can be used to semantically annotate the HTML representation of a thing’s RESTful API. Secondly, Microformats (as well as RDFa) are increasingly supported by search engines, such as Google and Yahoo, where it is used to enhance the search results. For example,

[20] http://www.w3.org/standards/semanticweb/

[21] http://www.w3.org/RDF/

[22] http://www.microformats.org

the “Geo” Microformat could be used to localise search results close to you or, in our context, to localise smart things in your direct vicinity.

More concretely, we use a compound of several microformats to describe our smart things. This helps the things to be searched by humans using traditional or dedicated search engines, but it also helps them being “discovered” and understood by software applications in order to automatically use them. As an example, in Figure 5.10 we use 5 microformats to describe a Sun SPOT and embed this semantic information directly in the HTML representation of the SPOT resources.

Fig. 5.10 Compound Microformats for Describing a Sun SPOT Using the Geo, hCard, hProduct and hReview Microformats

The listing shown in Figure 5.11 shows how to define the formal name (fn) of the Sun SPOT as well as an authoritative URL, where more information about the device can be found. We provide this semantic markup in the HTML representation of a Sun SPOT:

```
1     <span class=”fn”>Sun SPOT</span>
2     <span class=”URL>
3         <a href=”http://sunspotworld.com</a>
4     </span>
```

Fig. 5.11 Snippet of the HTML Representation of a Sun SPOT Including the hProduct Microformats

While there is still much research to be undertaken to be able to search for and discover smart things, the recent developments of the Web standards are going in the right direction for globally supporting such semantic descriptions. Indeed, a

derivative form of the already well supported Microformats, called Microdata,[23] might be part of the HTML 5 standard and might be widely adopted and understood by most next generation Web browsers and other Web clients.

5.5.3 Sharing Smart Things

The success of Web 2.0 Mashups depends on the trend for Web 2.0 service providers (e.g., Google, Twitter, Wordpress, etc.) to provide access to some of their services through relatively simple, often RESTful, open APIs on the Web. Mashup developers often share their Mashups on the Web and expose them through open APIs as well, making the service ecosystem grow with each application and Mashup. Figure 5.12 shows the simplified component architecture of a Social Access Controller (SAC), which serves as authentication proxy between clients and smart things.

To ensure the success of physical Mashups, they need to replicate the same level of openness. However, enabling such an open model for a Web of Things requires a sharing mechanism for physical things supporting access control to the RESTful services provided by devices. For example, one could share the energy consumption sensors in one's house with the community. However, this is a potentially risky process, given that these devices are part of our everyday life and their public sharing might result in serious privacy implications (if almost no energy has been used recently, the home owners may be on vacation and burglars might look for these kinds of patterns). HTTP already provides authentication mechanisms (e.g., HTTP Authentication[24]) based on credentials and server-managed user groups. While this solution is already available for free on most (embedded) Web servers, it still presents a number of drawbacks in the WoT context. First, for a large number of smart things it becomes quite unmanageable to share credentials for each of them. Then, as the shared resources are not advertised anywhere, sharing also requires the use of secondary channels, such as sending emails containing credentials to people. Several platforms, such as SenseWeb (Luo et al. 2008)or Pachube[25] propose to overcome these limitations by providing a central platform for people to share their sensor data. However, these approaches are based on a centralised data repository and are not designed to support decentralisation and direct interaction with smart things.

A promising solution is to leverage existing social structures of social networks (e.g., Facebook, Linkedin, Twitter, etc.) and their (open) APIs to share things. Using social networks enables users to share things with people they know and trust

[23] http://dev.w3.org/html5/md/

[24] http://www.ietf.org/rfc/rfc2617.txt

[25] http://www.pachube.com

(e.g., relatives, friends, colleagues, fellow researchers, etc.), without the need to recreate yet another social network or user database from scratch on a new online service. Additionally, this enables advertising and sharing through a unique channel: you can use various well-known social networks to inform your friends about the sensors you shared with them by automatically posting messages to their profile or newsfeed.

The SAC platform (Guinard et al. 2010a) is an implementation of this idea. SAC is an authentication proxy between clients (e.g., Web browsers) and smart things. Rather than maintaining its own database or list of trusted connections and credentials – as it would be done with simple HTTP authentication – SAC connects to a number of social networks (e.g., Twitter, Facebook, LinkedIn, etc.) to extract all potential users and groups one could share with.

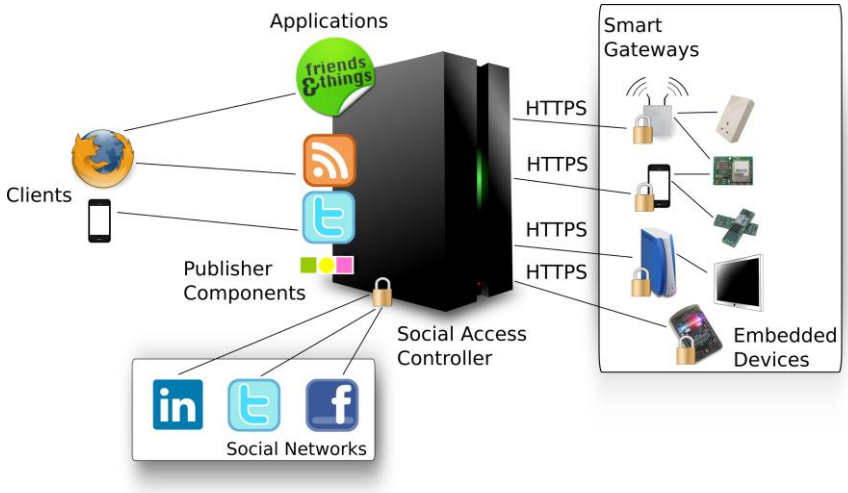


Fig. 5.12 Simplified Component Architecture of the SAC

This is possible as most social networks offer a Web API (e.g., Facebook Connect[26]). Providing an open Web API is one of the success factors of social networks themselves. Indeed, these APIs allow third-party Web applications to be built using partial data extracted from the social networks and thus to enhance the functionality of the social networks.

The sharing process occurs in three phases. First, the smart things owner accesses SAC by logging in, using at least one of his social networks credentials. SAC then uses delegated authentication with the social network to identify the owner. Afterwards, the smart thing to be shared has to be crawled in order to identify the re-

[26] http://developers.facebook.com/connect.php

sources and capabilities of its RESTful services, i.e., which functionalities can be shared for that thing. Finally, the user generates the access control list of the smart thing by selecting which friends can interact with what resource.

When an owner shares resources with a trusted connection, the latter is informed about it directly on their social network. In case of Facebook, it publishes a message to the news feed of the friend. In case of Twitter it simply tweets a message to the trusted connection (e.g., "Rachel shared her Ploggs Energy sensors with you"). The posted message also contains a *link* that redirects to the shared resource. The link does not point to the smart thing directly but to an instance of SAC that acts as the authentication proxy, as shown in Figure 5.12 When a trusted connection uses the provided link, SAC will verify its identity. If the friend is logged in successfully with one of their social networks, SAC will internally check whether this person also has access to the requested resource. If it is the case, SAC logs on the shared resource using the credentials provided by the owner when registering the resource. It then redirects the HTTP request of the trusted connection to the shared resource. Finally, it redirects the result directly to the HTTP client of the trusted connection, for example to a Web browser.

5.6 Discussing the Future Web of Things

Thanks to the wide availability of HTTP libraries and clients, and to the loose coupling, simplicity, and scalability properties of RESTful architectures, RESTful applications have rapidly become one of the most practical integration architectures. This makes it desirable to use Web standards for interacting with smart things. Although HTTP introduces a communication overhead and increases average response latency, it is still sufficient for many pervasive scenarios where longer delays do not affect user experience (Drytkiewicz et al. 2004; Priyantha et al. 2008). Previous work (Trifa et al. 2009; Yazar and Dunkels 2009) has shown that the performance of using HTTP as a data exchange protocol is largely sufficient for common pervasive scenarios, especially when only a few concurrent users are accessing the same resource simultaneously (200 ms mean response time with 100 concurrent users on a 1.1 GHz server running a Smart Gateway). We have also shown that caching techniques can significantly improve the performance of concurrent sensor data reading by using tools used for massively scalable Web sites (Trifa et al. 2009). These techniques can be directly applied to Web devices, given that devices have on-board HTTP support.

Web 2.0 Mashups have significantly lowered the entry barrier for the development of Web applications, which is now accessible to non-programmers. It should be noted that a resource-oriented approach should not be universally considered as the miracle solution for every problem. In particular, scenarios with very specific requirements, such as high performance real-time communications, might benefit

from tightly coupled systems based on different system architectures. However, for less constrained applications, where massive scalability, ad-hoc interaction, and serendipitous re-use are necessary, Web standards allow any device to speak the same language as other services on the Web. This makes the integration of the real-world with any other Web content much easier, so that physical things can be bookmarked, browsed, searched for, and used just like any other Web resource.

Based on our personal experience, the drawbacks of Web architectures are easily offset by a notable simplification of the application design, integration, and deployment processes (Guinard et al. 2009), in particular when comparing RESTful devices with other systems for embedded devices, such as WS-* Web services. As an example, the Plogg RESTful Gateway and the Sun SPOTs have been used by external development teams who read about our project on our Web site. In the first case, the idea was to build a mobile energy monitoring application based on the iPhone that communicates with the Ploggs. In the second case, the goal was to demonstrate the use of a browser-based JavaScript Mashup editor with real-world services. According to interviews we conducted with these developers, their experience confirmed ours. They enjoyed using the RESTful smart things, in particular the ease of use of a RESTful Web API versus a different kind of API. For the iPhone application, a native API to Bluetooth did not exist at that time. However, like for almost any platform an HTTP (and JSON) library was available. One of the developers mentioned a learning curve for REST but emphasised the fact that it was still rather simple and that once it was learnt, the same principles could be used to interact with a large number of services. They finally noted the direct integration to HTML and Web browsers as one of the most prevalent benefits.

5.7 Conclusion

In this chapter, we suggested that Web technologies are – contrary to popular belief – a suitable protocol for building applications on top of services offered by smart things. After summarising the core design principles of Web architecture, we proposed an architecture for the Web of Things based on the concepts of REST, syndication for smart things, Web Hooks, and Smart Gateways. We demonstrate the idea with several prototypes.

Thanks to the loose-coupling, simplicity and scalability of RESTful architectures, and the wide availability of HTTP libraries and clients, RESTful architectures are becoming one of the most ubiquitous and lightweight integration platforms. Because of this, using Web standards to interact with smart things seems to be increasingly adequate. Although HTTP introduces a communication overhead and increases average latency, it is sufficient for many pervasive scenarios where such longer delays do not affect user experience.

Introducing support for Web standards at the device-level is beneficial for developing a new generation of networked devices that are much simpler to deploy, program, and reuse. Applying the same design principles that supported the success of the Web, in particular openness, connectedness, and simplicity, can significantly leverage the ubiquity and versatility of the Web as a common ground for supporting interactions between devices and applications. Furthermore, as most mobile devices have already Web connectivity and Web browsers, and most programming environments support HTTP, we tap into the very large Web developer community as potential application developers for the Web of Things.

References

Drytkiewicz W, Radusch I, Arbanowski S, Popescu-Zeletin R (2004) pREST: a REST-based protocol for pervasive systems. Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems

Duquennoy S, Grimaud G, Vandewalle J (2009) The Web of Things: interconnecting devices with high usability and performance. Proceedings of the 6th IEEE International Conference on Embedded Software and Systems (ICESS'09). HangZhou, Zhejiang, China

Fielding RT (2000), Architectural styles and the design of network-based software architectures. Ph.D. Thesis, University of California. Irvine, USA

Floerkemeier C, Lampe M, Roduner C (2007) Facilitating RFID Development with the Accada Prototyping Platform. Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops. IEEE Computer Society

Guinard D, Trifa V, Pham T, Liechti O (2009) Towards Physical Mashups in the Web of Things. Proc. of the 6th International Conference on Networked Sensing Systems (INSS). Pittsburgh, USA

Guinard D, Fischer M, Trifa V (2010a) Sharing Using Social Networks in a Composable Web of Things. Proceedings of the 1st IEEE International Workshop on the Web of Things (WoT 2010) at IEEE PerCom, Mannheim, Germany

Guinard D, Mueller M, Pasquier J (2010b) Giving RFID a REST: Building a Web-Enabled EPCIS. Proceedings of the IEEE International Conference on the Internet of Things (IOT 2010). Tokyo, Japan

Guinard D, Trifa V, Wilde E (2010c) A Resource Oriented Architecture for the Web of Things. Proceedings of IoT 2010, IEEE International Conference on the Internet of Things. Tokyo, Japan

Guinard D, Trifa M, Karnouskos S, Spiess P, Savio D (2010d) Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services., IEEE Transactions on Services Computing. 3, 223–235

Hui J, Culler D (2008) Extending IP to low-power, wireless personal area networks. IEEE Internet Comput 12:37-45

Hui J, Culler D (2008) IP is dead, long live IP for wireless sensor networks. Proceedings of the 6th ACM conference on embedded network sensor systems. ACM, Raleigh, NC, USA

Kindberg T, Barton J, Morgan J, Becker G, Caswell D, Debaty P, Gopal G, Frid M, Krishnan V, Morris H, Schettino J, Serra B, Spasojevic M (2002) People, places, things: web presence for the real world. Mob Netw Appl 7:365-376

Luckenbach T, Gober P, Arbanowski S, Kotsopoulos A, Kim K (2005) TinyREST - A protocol for integrating sensor networks into the internet. Proceedings of the Workshop on Real-World Wireless Sensor Network: SICS. Stockholm, Sweden

Luo L, Kansal A, Nath S, Zhao F (2008) Sharing and exploring sensor streams over geocentric interfaces. Proceedings of the 16th ACM SIGSPATIAL international conference on advances in geographic information systems. ACM, Irvine, California

Pautasso C, Wilde E (2009) Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. Proceedings of the 18th International World Wide Web Conference (WWW2009). Madrid, Spain

Priyantha NB, Kansal A, Goraczko M, Zhao F (2008) Tiny web services: design and implementation of interoperable and evolvable sensor networks. Proceedings of the 6th ACM conference on embedded network sensor systems. ACM, Raleigh, NC, USA

Richardson L, Ruby S (2007) RESTful Web Services. O'Reilly Media, Inc

Stirbu V (2008) Towards a RESTful Plug and Play Experience in the Web of Things. Proceedings of the IEEE International Conference on Semantic Computing

Trifa V, Wieland S, Guinard D, Bohnert TM (2009) Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices. Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE 09). Marina del Rey, CA, USA

Trifa V, Guinard D, Davidovski V, Kamilaris A, Delchev I (2010) Web-based Messaging Mechanisms for Open and Scalable Distributed Sensing Applications. Proceedings of the 10th International Conference on Web Engineering (ICWE 2010). Vienna, Austria

Wilde E (2007) Putting Things to REST. School of Information. UC Berkeley

Yazar D, Dunkels A (2009) Efficient Application Integration in IP-based Sensor Networks. Proceedings of ACM BuildSys, the First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings, BuildSys. Berkeley, USA

Yu J, Benatallah B, Casati F, Daniel F (2008) Understanding Mashup Development. IEEE Internet Comput 12:44-52

Zang N, Rosson MB, Nasser V (2008) Mashups: who? what? why?. Proceedings of CHI '08 extended abstracts on Human factors in computing systems. ACM, Florence, Italy

Index Chapter 5