Thus far, we have discussed the use of the IP architecture as the means by which smart objects are connected. We have discussed how the IP architecture is built, how IP works with message routing, and how IPv6 fits the requirements for smart objects exceptionally well. But we have not yet discussed how these technologies are used to create smart object systems, and how smart objects can be integrated into existing IT systems. In this chapter, we take a look at web services - a technology by which smart objects can be efficiently integrated into existing IT and enterprise business systems.

Web services are a framework for building distributed applications. They have typically been used to build applications that either interact using a web browser, or are somehow related to the World Wide Web. But the technology that makes up web services is not tied to the World Wide Web or the particular technology that typically is associated with it, such as web browsers.

Web services are typically explained using examples from popular services that are used on the Web, such as flight travel booking systems, online book stores, or web searches. Even though those applications may sound removed from smart object systems, their inherent application properties are surprisingly similar to those in smart object systems. Web-style applications and smart object applications share many of the basic communication properties: they are composed of separate systems that exchange data.

Given the prevalence of the Web and its associated technologies, web services have seen a
tremendous adoption in the general purpose IT world in the past couple of years. All major programming languages provide libraries tailored to build web-service-oriented applications. Hence, a large body of existing IT systems is built using web services. There are numerous online courses and other training material available to learn how to build web service applications.

Web services have traditionally been seen as a technology suitable for big servers, big datasets, and big systems. This technology has been used to couple database systems with each other in a framework that permits an expression of high-level concepts and dependencies, and yet is succinct enough to be standardized across a wide range of applications.

By using web service technology for smart object applications, existing web-service-oriented systems, programming libraries, and knowledge can be directly applied to the emerging field of smart object applications. This provides several benefits.

For businesses, smart object applications can be directly integrated with existing business systems and use the same interfaces and systems existing business systems use. This makes it possible to integrate smart object applications into enterprise resource planning systems without any intermediaries, thus reducing the complexity of the system as a whole. For industries, smart object applications can be built using off-the-shelf technology without any customized interfaces or translators. Systems can be built without requiring smart object specialists in every step of the project.

In this chapter, we discuss the use of web service technology for smart objects. Because of the expressiveness of the underlying principles, web services are highly suitable for smart objects. Despite the dominating belief that web services are a heavyweight concept, we demonstrate that they are indeed lightweight enough to be used for the resource-challenged environment in which smart objects exist. We do not discuss the details of web services in this chapter, however, as the concepts and the surrounding technology and its software are extensive and diverse. We keep the discussion at a relatively high level, and refer to more specialized publications for further details.

We examine the technology and principles behind web services, how they map onto smart object concepts, and how they can be efficiently implemented for smart objects. To ground the discussion, a concrete example of an existing web service for smart objects is provided: the Pachube service is a data-hosting service for smart-object-style applications where data are inserted and accessed using web service technology.

The performance of web services for smart objects has been questioned, because web services were initially used for large server systems. At the end of this chapter, we critically examine this by discussing the performance of published web service implementations for smart object systems. We find that the performance of web services for smart objects is indeed reasonable.

**Web Service Concepts**

**9.1 WEB SERVICE CONCEPTS**

Web services are typically described as communication between business servers, typically initiated by the interaction of a user through a web site. We make no exception to this, because this not only typifies the behavior of the web service technology, but also highlights the machine-to-machine communication aspects of web services.
Figure 9.1 is a canonical example of web service technology in action. In this figure a user is interacting with the website of a travel agent through a web browser. Although this part uses the World Wide Web, it is not where web services are used. Rather, web services enter where the interaction between the user and the website ends.

![Diagram of a web service transaction](image)

**FIGURE 9.1** A traditional example of a web service transaction. A human user uses the travel agent's website to look for flights. The travel agent server uses web service technology to query multiple airlines for possible routes and gets several results.

Web services are used as a communication mechanism between the travel agent's server and other servers that it communicates with to achieve its task. Neither the travel agent's server nor the airlines' servers are run by the same parties, so web services provide an intermediary that allows them to exchange data without in-between translation, since all servers adhere to the web services framework.

The travel agent's server sends a series of web service requests to a number of servers that belong to airline companies. The travel agent server receives replies from a number of them; in this case one from SAS and one from KLM. The travel agent server can then format these replies and present them to the user. The presentation of the data is independent of the format in which the data were sent between the airline servers and the agent server.

The example in Figure 9.1 illustrates how web services work, but not how they relate to smart objects. To shift the focus back to smart objects, Figure 9.2 shows an example of a smart object system realized through web services.
FIGURE 9.2 A smart object system implemented with web services. Temperature sensors post temperature data to a building automation server. The building automation server configures a radiator based on the temperature data.

In this example, a building automation server is connected to a network of temperature sensors and radiators. The sensors and radiators are located in a building and connected either to a wireless network or a wired in-house network. The particular communication technology used does not matter, because the system is built on IP.

The temperature sensors periodically post their temperature data to the building automation server using a web services framework. This allows the building automation server and the temperature sensors to be provided by different vendors since they both agree on a common communication mechanism and data format. The server may also query the sensors to get the current temperature value.

After the building automation server has received the temperature data from the house’s temperature sensors, it uses this information to control the radiator. The radiator has a smart object with web service communication abilities, and the building automation server posts a configuration request to the radiator. The radiator updates its setting to match the value requested by the automation server.

```xml
<xml>
  <sensors>
    <sensor>
      <name>Temperature</name>
      <value>27.1</value>
    </sensor>
  </sensors>
</xml>
```

FIGURE 9.3 An XML document.
Common Data Formats

9.1.1 Common Data Formats

Web services are a mechanism for exchanging data between disparate systems that are not developed by the same parties. The systems may be developed in different programming languages and run on vastly different hardware, but they still wish to exchange data in a system-independent way. For this reason, system-independent data formats are central to the web services framework.

The Extensible Markup Language (XML) is the most common data format associated with web services, but it is not the only data format available. In fact, the web services framework is not dependent on any particular data format, because it can operate across a range of data formats.

XML is a general purpose document format that provides a structured mechanism to encode machine-readable information. In addition to being machine-readable, XML documents are also human-readable, making them readily created and edited by humans as well as machines.

XML documents are composed of a set of tags, where each tag is shown as the name of the tag enclosed in the < and > characters. A tag consists of an open tag and a close tag. The close tag is the same as the open tag, but with the /character in front of the name. An example of a tag is <data>one</data>, where <data> opens the tag and </data> closes the tag. The value of the tag is the text between the open and close tag, which in this case is "one". Tags are nested to form a tree of tags.

An example of an XML document is shown in Figure 9.3. This XML document contains a temperature value from a fictional temperature sensor. The name of the sensor, "Temperature", is given in the <name> tag and the value is given in the <value> tag. The <sensor> tag contains information about one particular sensor. In this document, only a single sensor is present but more can be added following the closing of the <sensor> tag.

Because XML is a relatively verbose format, several ways to compress XML have been explored. The structured nature of XML makes XML possible to compress using source-specific techniques, and a number of variants of XML that use binary encodings rather than textual representations of the XML tags exist. There is, however, no standard for binary XML and none of the available formats have succeeded in achieving a de facto standard status.

An alternative to the verbose XML format is the JavaScript Object Notation (JSON) format. JSON is specified in RFC4627 [45] and provides a more lightweight markup than XML. Although JSON originally was designed to be easy to parse by JavaScript programs, the format is independent of any particular programming language. Libraries for parsing and constructing JSON messages are available for most programming languages.
An example of a JSON document is shown in Figure 9.4. This document contains the same information as the XML document in Figure 9.3, but is much more compact. The markup is more lightweight, but provides less means by which the document can be automatically translated between different formats. Yet, the JSON format is a good match for smart object systems, where compactness of representation is important due to the inherent resource constraints.

```
{"sensors": [
  {
    "name": "Temperature",
    "value": 26.1
  }
]
```

**FIGURE 9.4** A JSON document.

**Representational State Transfer**

**9.1.2 Representational State Transfer**

There are several ways to realize the web service concept. Some realizations are built on mechanisms that require significant processing power and communication bandwidth, whereas others are more lightweight.

In this discussion, we do not go into detail about the web service mechanisms on the expensive side of the spectrum. Examples of mechanisms that we do not discuss are SOAP (originally defined as Simple Object Access Protocol, but is now known as only an acronym), the Web Services Description Language (WSDL), and the Universal Discovery Description and Integration mechanism (UDDI). For the description of those standards and mechanisms, please see the standards documents or the many online descriptions available.

The resource constraints inherent in smart objects regarding processing power, energy, and communication bandwidth necessitate the use of lightweight mechanisms. Despite a reputation as a heavyweight, web services have nothing inherent in their interaction models, communication mechanisms, or concepts that make them heavyweight.

Representational state transfer (REST) is a lightweight instantiation of the web services concept that is particularly well suited to the properties of smart objects. REST is not just a web service instantiation, but an architectural model for how distributed applications are built. Systems built around the REST architecture are said to be RESTful.

REST builds on three concepts: representation, state, and transfer:
• Representation: Data or resources are encoded as representations of the data or the resource. These representations are transferred between clients and servers. One example of a representation of a resource is a temperature value written as a decimal number, where the representation is the decimal number and the temperature is the resource.

• State: All of the necessary states needed to complete a request must be provided with the request. The clients and servers are inherently stateless. A client cannot rely on any state to be stored in the server, and the server cannot rely on any state stored in the client. This does not, however, pertain to the data stored by servers or clients, only to the connection state needed to complete transactions.

• Transfer: The representations and the state can be transferred between client and servers.

REST, as an architectural model, describes the interactions we have seen so far in this chapter. In the example of the travel agent (Figure 9.1), the request for a reservation between Stockholm and Tokyo was a representation as were the replies from the airline servers. Likewise, the building automation system illustrated in Figure 9.2 contains temperature data and radiator configurations as representations.

REST is an architectural model that can be efficiently implemented as a combination of the Hypertext Transfer Protocol (HTTP) [83] and TCP/IP. With this instantiation of REST, HTTP requests are used to transfer representations of resources between clients and servers. Uniform Resource Identifiers (URIs) are used to encode transaction states.

With this implementation of the REST architecture in mind, we return to the building automation example in Figure 9.2. In this example, the temperature sensors submit their temperature data to the building automation server using the HTTP PUT method. To query sensors, the server uses the HTTP GET method. The server then sends its configuration request to the radiator using the HTTP PUT method.

An example REST HTTP transaction for smart objects
To make the discussion concrete, we turn to a detailed discussion of how a REST HTTP transaction for smart objects looks. Using the transaction from the building automation system in Figure 9.2, we focus on the server’s request for temperature data from one of the sensors. This request is implemented by using the HTTP GET request, which is issued by the server to
one of the sensors. The sensor responds with the temperature data of the sensor in JSON format.

The HTTP GET request sent by the server is shown in Figure 9.5.

```
GET /sensors/temperature HTTP/1.1
Content-type: application/json
```

**FIGURE 9.5** An HTTP GET request for the data of a temperature sensor in JSON format. The server's response is shown in Figure 9.6.

The HTTP request, which is human-readable, consists of two lines of text. The first line contains the HTTP GET verb, followed by the URI that represents the temperature sensor. In this case this is as simple as /sensors/temperature, but more complex URIs are possible. Ending the first line is the name and version of the HTTP protocol. HTTP 1.1 is the current version of the HTTP protocol.

The second line of the server's request contains the requested representation of the data that the client has to offer. This line contains the HTTP header "Content-type" followed by the type "application/json". This type is defined in the JSON specification as the content type to be used for JSON data [45].

The client's response to the server's request is shown in Figure 9.6.

```
HTTP/1.1 200 OK
Content-type: application/json
{"sensors": [{"name": "Temperature", "value": 26.1}]}
```

**FIGURE 9.6** HTTP response for the HTTP request in Figure 9.5.

Again, this HTTP reply is in a human-readable format. The reply consists of two parts, the HTTP header and the HTTP body. The
header is two lines long. The first line contains the HTTP/1.1 keyword, which again tells the receiver that this reply is in HTTP version 1.1 format. This keyword is followed by the HTTP status code 200, which tells the receiver that the HTTP request was successfully processed. The “OK” following the status code is a human-readable representation of the status code.

The HTTP reply contains the same "Content-type" header as the request, which tells the receiver that the data in the HTTP body are in JSON format. Following the HTTP header is a blank line that divides the header from the body.

The HTTP body contains the JSON data that represent the current temperature as sensed by the smart object’s sensor.

This HTTP request is transported over a TCP connection, as discussed in Chapter 7. Armed with the knowledge of how TCP works, we can now construct a detailed picture of how the entire REST transaction between the building automation server and the smart object client looks.

Figure 9.7 shows the full REST transaction including all packets that are sent for the complete transaction. The transaction is divided into three phases: the TCP connection open phase, the REST data transfer phase, and the TCP connection close phase.

**FIGURE 9.7** A full REST web service transfer over HTTP/TCP/IP with all packets indicated.

The transaction starts with the TCP connection phase. The TCP
connection is opened by the exchange of the TCP SYN and TCP SYNACK segments between the server and the client. The server sends the TCP SYN to the client, and the client responds with the TCP SYNACK. When the server has received the TCP SYNACK segment, the connection enters the REST data transfer phase.

During the REST data transfer phase, the server sends the HTTP request from Figure 9.5 as one or more TCP segments to the client. The HTTP request may fit in a single TCP segment, if the segment size is large enough. The segment size for a connection is determined by the client and server during the setup of the TCP connection. In this case, however, the HTTP request is small enough to fit into most TCP segment sizes. All TCP segments are acknowledged by the receiver, and the sender retransmits the segments if it does not receive the acknowledgments. In this case, however, the acknowledgments are piggybacked on the data packets sent in response to the reply.

The client responds to the request with the response from Figure 9.6. Again, this is sent as one or more TCP segments, depending on the size of the response and on the maximum segment size of the connection. The data packets are acknowledged. If the entire HTTP response fits in a single TCP segment, the acknowledgment for the data is piggybacked on the TCP FIN segment sent during the TCP connection close phase.

After the data transfer is complete, both the server and the client close the connection by sending a TCP FIN segment to the peer. This segment is acknowledged by a final TCP ACK segment, and the entire transaction is complete.

Coming up in Part 2: The performance of web services for smart objects.

Printed with permission from Morgan Kauffman, a division of Elsevier. Copyright 2010. "Interconnecting Smart Objects with IP: The Next Internet" by Jean-Philippe Vasseur and Adam Dunkels. For more information about this title and other similar books, please visit www.elsevierdirect.com.

References:

Related links:
Rise of the Embedded Internet