16th Conference on Water Distribution System Analysis, WDSA 2014

Web Services for Water Systems: the iWIDGET REST API

M.G. Barry* a, M.E. Purcell a, B.J. Eck a, J. Hayes a, E. Arandia a

a IBM Research, Dublin, Ireland

Abstract

The iWIDGET project has a special focus on the role of smart water meters for understating consumption and improving system operations. To support the project and future commercial developments, a distributed platform supported by a web based application programming interface (API) has been developed to allow developers to rapidly prototype and develop applications and user interfaces for different devices and systems including Windows, Mac OS, Linux, iOS, Android and others. The iWIDGET API is based on representational state transfer (REST), with data accessed by a web address or URI structured as a request or query.

1. Introduction

Smart water meters — meters that can automatically collect detailed usage data — are growing in popularity as a way to manage aging infrastructure and encourage conservation. Smart meters are a new source of information about water systems, providing consumption data in greater spatial and temporal detail than ever before. In order for consumers and utilities to take advantage of smart meter technology, the iWIDGET project (www.i-widget.eu) is developing a suite of applications including usage evaluation, demand forecasting, pump scheduling, water balance estimation and many others. Collectively, these applications aim to develop novel, robust, practical and cost-effective methodologies and tools to manage urban water demand by reducing wastage, by improving the utilities’ understanding of end-user demand, and by reducing customer water and energy costs [1].

Supporting the desired suite of applications for consumers, utilities and other stakeholders requires consistency in information access, interoperability between applications, and flexibility in implementation. To satisfy these requirements, iWIDGET is a distributed architecture [2]. In summary, the system is composed of an iWIDGET server and database, with remote client systems for data acquisition, analysis (Fig. 1) and visualization. Data acquisition from meters in the field is performed by a water utility. A data acquisition service is used to input smart meter readings into the iWIDGET system on a regular schedule. Analytic functions are delegated to separate, remote analytic servers.

* Corresponding author. Tel.: +353-1-826-9185
E-mail address: michael.barry@ie.ibm.com
Analytics obtain data from the iWIDGET server and possibly other sources, perform analysis, and return results. Analytic results can be visualised as graphs, tables or other formats. Visualisation is realised through interactive applications, for example web portlets.

Communication between iWIDGET components and interaction with external users is supported through the iWIDGET Application Programming Interface (API). The iWIDGET API is a set of functions through which users, applications and analytics can interact with the iWIDGET server and can access and manipulate data. The API is implemented using web services [3].

The remainder of this paper provides an overview of the iWIDGET API, its implementation and usage. The next section presents different elements in the API and how they interact with each other. Sections 3 and 4 describe the role of web services and how the iWIDGET API is implemented using RESTful concepts and techniques. Section 5 illustrates the use of the API by an example analytic.

2. iWIDGET Conceptual Object Model

The iWIDGET system utilises web services that allow users and programs in different locations to retrieve data using web addresses. The structure or pattern of the web address forms a complete request for the server. Structures of the web address (URIs) follow from the conceptual object model explained here. This conceptual object model provides a way of thinking about the world that reflects the iWIDGET emphasis on data retrieval from smart meters. Five distinct objects comprise the model:

- time series
- device
- query
- result
- data store

Time series are collections of values each associated with a time. An individual time series may contain values of raw data from field measurements or values that result from calculation. A time series may have a regular or irregular time step. Time series are described by a type, physical unit and period. For example a time series of values from a flow meter could have a type of volumetric flow rate, physical unit of litres per minute and period of 1 hour. The relationship between time series is provided by devices.

A device is a reporting point for time series of values. Devices represent a level of aggregation ranging from none, in the case of an individual sensor, to a high level of aggregation such as a city. The concept of a device applies to real physical sensors as well as to the environment in which those sensors reside. Examples of devices include:

- individual consumption points, where water usage is measured;
- inlet and outlet points to District Metered Areas (DMAs), where both flow and pressure are reported; and,
- cities, where aggregated information such as total usage or average night flow is calculated.
Each device has a type, parent device, and optionally, several time series. The parent device supports a hierarchical organization of devices to reflect the underlying structure of the system. For example, an individual consumption device could reside within an apartment building, which resides inside a DMA, which resides in a city. Fig. 2 illustrates the relationship between devices and time series.

The query, results and data store objects support analysis, visualization, and interaction on the iWIDGET system. The analysis components or analytics of the iWIDGET architecture answer questions that are formulated as a query. Queries can be generated interactively by a user, triggered automatically, or generated by another analytic. The analysis specified by a query may require the analytic component to access devices, time series and data stores on the system or to update these. Data stores provide supplemental knowledge about the characteristics of the metering system under consideration, such as topology, geographical maps, or data about electricity pricing tariffs.

The answer to a query is returned as a result. A result is summary of the outcome of an analytic. Queries and results are mapped to an analytic component and the format of queries and results is unique to each analytic. Analytics interact with queries, results, time series and data stores as shown in Fig. 3.

Visualization engines interact with API components to present information to the user. Visualization may rely on times series, devices or results in the system. There is therefore a close coupling between the format of results generated by an analytic, such as a pump schedule, and how those results are visualized. Analysis and visualization tasks are independent, allowing for many deployment scenarios. iWIDGET does not support any individual visualization engine or technique. Section 5 below presents an example User Interface (UI) application that visualises the results of an adaptive pricing analysis.

The model presented here evolved 'from the ground up' through an iterative (agile) process aimed at supporting applications under active development and test within the project. The ∼20 use cases supported so far suggest that the conceptual model will be useful for additional applications. Further details on the implementation of this model as a RESTful web service are described in the following sections.

3. Web Services in iWIDGET

In the iWIDGET architecture, web services connect the data storage, analysis, and visualization components. A web service is a method of communication that allows two software systems to exchange data over the internet in a platform and programming language independent manner [4]. Individual web services are software functions provided at a network address [5]. Several types of web services exist including Simple Object Access Protocol (SOAP) and Representational State Transfer (REST). The iWIDGET architecture uses RESTful web services [6], providing a fast, powerful platform for rapid prototyping and implementation of new distributed web-based applications; as well as supporting easy integration of existing tools and services.
RESTful web services adhere to the architectural principles of representational state transfer. As the name suggests, only a representation of the resource state is transferred and not the state itself. The principles of a RESTful architecture require a uniform interface between clients and servers and encapsulated requests. Resources (data) are identified by web address and manipulated using HTTP operations: GET, PUT, DELETE, POST. Data are represented using an internet media type. Further detail on the architectural principles of REST and RESTful web services are beyond the scope of this paper; the interested reader is referred to the references, particularly [6].

In the iWIDGET implementation of RESTful web services, the iWIDGET server hosts data while clients initiate requests to the server. Client requests may access data for analysis or visualization. Requests may also add new data to the system for later use. The server returns data represented in JavaScript Object Notation (JSON) [7]. JSON is an internet media type that is lightweight and human-readable. The iWIDGET server can also return data in alternative formats including WaterML [8]. The request/response mechanism and use of JSON and WaterML are ideally suited to the simple, fast, self-contained exchange of resources of a RESTful implementation.

4. iWIDGET API Walkthrough

The API provides a set of functions, accessible as REST URIs to support access to and modification of devices, time series, data stores and user queries and results. This section illustrates selected API features. A complete version of the API will be published in forthcoming iWIDGET deliverables.

4.1. Devices

Devices are identified by a device identifier that maps to some unique identifier assigned by the water utility such as the serial number of the device. Information about individual devices can be retrieved using a URI of the form iWidget/device/{deviceID}. The result will look similar to the REST request/response pair shown in Fig. 4.

The response includes the device identifier, device type, parent, and customer type. Devices can also be selected on a cross-network, or sub-network basis according to type. Devices (including parent devices) can be either physical or virtual as described in section 2 above. A parent device can be used to select a sub-tree of the network using the URI iWidget/deviceTree/{rootID}. Devices can also be identified and sorted according to customer type, e.g. residential, commercial, hospital.

Repeated API calls to iWidget/device/ and to its companion URIs allow the user to construct a map of the network based on the relationships between devices, device types and categories of customer. Fig. 5 illustrates how a network map can be rendered for the user using a tree or a drop-down box. A partial set of URIs for devices is shown in Fig. 6.

4.2. Time Series

A time series is identified by type and device, as illustrated in Fig. 7. Period and units can be appended to the REST URI as optional parameters. A request for time series data also includes a start time and an end time for the series specified as HTTP query parameters. The response contains the device identifier, a time series containing a set of date, value pairs for the time selected, the units of the values, and the period of the time series.

Time series values are added to the system using a REST POST request. Data to be added is described as a JSON object as depicted in Fig. 8. The JSON object contains details on the value and type of the data being added, the
Fig. 5. Depicting devices using a tree or as a drop-down list

GET iWidget/devices
GET iWidget/deviceTypes
GET iWidget/devices/{deviceType}
GET iWidget/deviceTree/{rootID}
GET iWidget/deviceTree/{rootID}/{deviceType}
GET iWidget/device/{deviceID}
GET iWidget/device/{deviceID}/type
GET iWidget/device/{deviceID}/customerType

Fig. 6. iWIDGET API for retrieving Device Information

GET: http://www.iWidgetServer.net/iWidget/timeSeries/00056868/flowData
?startTime=2009-03-01 03:00:00&endTime=2009-03-01 04:00:00
Response: ["deviceID":"00056868", "units":"m3/s", "period":"hourly", "timeSeries":[{"value":"1.54469","date":"2009-03-01 03:00:00"},
{"value":"1.5450001","date":"2009-03-01 04:00:00"}]

Fig. 7. iWIDGET REST call and iWIDGET JSON encodings for a measurement time series

POST http://www.iWidgetServer.net/timeSeries/leakDetector
Data: {"deviceID":"Block1","type":"Leak","period":"Week","value":0.1,"date":"2012-01-01","units":"m3/s"}

Fig. 8. iWIDGET REST call to post a value to a time series

date it was calculated and the period of the measurement. The identity of the analytic widget posting the measure is tracked from the URI. In the example the leakDetector analytic is generating and posting the measurement.

Companion URIs for measurement series allow different types of data, e.g. volume series, pressure data to be selected. A list of devices with different parents can also be selected. Fig. 9 depicts a partial set of RESTful calls for manipulating timeSeries data.
4.3. User Queries and Results

The iWIDGET API mediates communication between components using queries and results. This arrangement recognizes that analytics may not generate a result in a user-friendly response time. It also enforces the loose coupling between the system components as required by the system architecture. This loose coupling allows iWIDGET components to use a range of implementation, execution, and management techniques. The primary API for the exchange of queries and results are the iWidget/query, iWidget/queries and the iWidget/results REST URIs. The procedure for a user application to trigger an analytic is outlined in Fig. 10.

First the user application gathers any necessary input data, and then posts a query for the intended analytic to the iWIDGET server, via the API. Analytics constantly poll the iWIDGET server for new requests. Upon notification of one or more new queries, the analytic fetches individual queries using a GET call on the REST API. Having received the query and the inputs for the analytic, the analytic executes.

Once the analytic is finished, it posts a result for the query to the server. The user application polls the iWIDGET server to detect when new results are available. Front end user applications fetch new results for presentation to the end-user. Other types of user applications can automatically fetch the result for further processing. When the user application is finished with the result it marks the query as closed using a REST PUT procedure. This prevents previously handled queries from being accidentally re-processed by the analytic.

5. Example Adaptive Pricing

The adaptive pricing analytic is used to illustrate how the iWIDGET API can support implementation using a variety of underlying technologies. Adaptive pricing provides a set of algorithms that allow a network provider or utility, to explore and dynamically set prices for water. Several tariff schemes are considered including time of use, seasonal use and peak-time rebates.
The analytic calculates the time period where consumption is higher, for both weekdays and weekends, and generates a suggested price schedule aiming to shift the peak demand.

The iWIDGET solution for adaptive pricing consists of UI and analytic parts. From the UI perspective, the solution is a 3-layer architecture, as shown in Fig. 11, consisting of a presentation tier, a business logic tier and a data tier. The presentation tier is where users interact with the application. In the adaptive pricing widget the presentation tier is a JSR 168 portlet running on an IBM WebSphere® Portal server and accessed by the user using a web browser. The business logic tier is the REST API, that runs on an application server and mediates the communication between the user and the data tier. The data tier is the iWIDGET Database.

The portlet itself is written in JavaScript [9], which provides a number of tools and toolkits, such as dojo [10], for manipulating data and graphics. In addition JavaScript/dojo includes very good support for interacting with REST APIs. The aim of the portlet is to allow the user to construct a query that can be processed by the adaptive pricing analytic. The user selects:

1. The devices under consideration from a list populated using the REST API,
2. the algorithm to be used; and,
3. a range of dates for the algorithm to act on.

The analytic application runs on a remote system as depicted in Fig. 1. The adaptive pricing analytic is written in R and uses the RCurl package to interact with the iWIDGET Server using the REST API. The application fetches queries from the iWIDGET server as illustrated in Fig. 10. It then retrieves the time series information necessary for the analytic using the iWIDGET time series set of URIs described in Fig. 9. Using the time series data a pricing plan is derived. Once the analytic is complete it posts a result to the iWIDGET server as outlined in Fig. 8. New results are picked up by the adaptive pricing UI, rendered using the JavaScript/dojo tools and presented to the user. A sample output including demand averages and a pricing plan is shown in Fig. 12.

6. Conclusions

This paper has presented an application programming interface that serves as a platform for the analysis of time series data. The API is based on a conceptual model of five objects. These objects are used as the basis for building universal resource identifiers. In accordance with RESTful principles, applications on the platform access and manipulate data by these URIs through a web service. Using these services, developers can write applications and user interfaces for different devices and operating systems including Windows, Mac OS, Linux, iOS, Android and others.

This arrangement has several advantages over existing approaches, and opens many new possibilities for households, water utilities, engineering firms, meter manufactures and others. With proper permissions, households can access their usage data. Developers can write applications suited for any water system where data is provided by the iWIDGET API. Utilities can make data available, openly or otherwise, in a standard format. Existing utility databases can support an iWIDGET deployment by implementing the API rather than duplicating data into a new system. Engineering or planning firms can develop custom tools to re-use on subsequent projects.
The iWIDGET API, based on RESTful services, supports rapid prototyping and implementation of water analytics. It is hoped that the iWIDGET API will become a standard way of working with data on water systems.

Acknowledgements

Acknowledgement: The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318272

References

[1] iWIDGET, Improved water efficiency through ict technologies for integrated supply-demand side management, 2012. URL: http://cordis.europa.eu/projects/rcn/105822_en.
[2] M. E. Purcell, M. G. Barry, B. J. Eck, Using smart meters in (near) real-time on the iwidget system, Proceedings of the 11th International Conference on Hydroinformatics, New York (2014).
[3] R. T. Fielding, R. N. Taylor, Principled design of the modern web architecture, ACM Transactions on Internet Technology 2 (2002) 115–150.
[4] R. T. F. et al., Hypertext Transfer Protocol, HTTP/1.1, Technical Report, Internet Engineering Task Force, 1999. RFC 2616.
[5] E. Cerami, Web Services Essentials, O’Reilly & Associates, Inc., 2002.
[6] R. T. Fielding, REST: Architectural Styles and the Design of Network-based Software Architectures, Ph.D. thesis, University of California, Irvine, 2000.
[7] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, Technical Report, Internet Engineering Task Force, 2014. RFC 7159.
[8] P. Taylor, WaterML 2.0: Part 1 – Timeseries, Technical Report, Open GeoSpatial Consortium, 2014. Ref: 10-126r4.
[9] D. Flanagan, JavaScript - the definitive guide: activate your Web pages: covers JavaScript 1.2 (3. ed.), O’Reilly, 1998.
[10] J. E. Harmon, Dojo using the Dojo JavaScript library to build Ajax applications, Addison-Wesley, 2009.