HOBOlink® Web Services V2 Developer's Guide

Onset Computer Corporation
470 MacArthur Blvd.
Bourne, MA 02532
www.onsetcomp.com

Mailing Address:
P.O. Box 3450
Pocasset, MA 02559-3450

Phone: 1-800-LOGGERS (1-800-564-4377) or 508-759-9500
Fax: 508-759-9100
Email: loggerhelp@onsetcomp.com

Technical Support Hours: 8AM to 8PM ET, Monday through Friday
Customer Service Hours: 8AM to 5PM ET, Monday through Friday

17382-D

Contents

Section 1: Overview

HOBOlink Web Services offer the ability for third-party developers and partners to write their own programs to extract HOBO® U30 data from the HOBOlink database.

HOBOlink exposes a set of SOAP web services hosted on Onset's remote servers. These web services are accessed through a third party software program (the "caller"). In addition, you can also access public and private devices through Onset REST web services, covered in Section 5.

The SOAP Web Services deliver customized datasets to the caller in a known XML format (SensorML) with the interface to the data (WSDL file) published at a known URL and the XML schema published by Onset. This relies on HOBOlink's underlying infrastructure where data is stored as individual readings, such that custom datasets can be extracted and sent to the caller.

Specific capabilities of HOBOlink SOAP Web Services include:

- Data from the logger is stored as individual entries in a data warehouse type of database (SOS server), rather than only in binary .dtf files. This provides a tremendous amount of extensibility for allowing customized access to a user's data, both public and private.
- Data is streamed to the caller in SensorML, an industry standard XML format. Publishing the XML schema allows developers to use any number of data binding tools that allow manipulation of the XML document using simple programming objects.
- Datasets returned to the caller can span multiple devices, launches, and wraps, as well as being constrained to only a small subset of a deployment.
- Datasets are customizable by time, device S/N, sensor S/N and/or measurement type.
- Authentication is achieved via a token (provided by Onset) that is passed to HOBOlink as part of the web services call. HOBOlink manages access to the various web services using these tokens. Tokens will be provided free to customers who purchase HOBO U30 devices.

If you wish to discuss The HOBO Remote Monitoring System and Web Services in more detail, please contact an Onset Computer Application Specialist at (800) 564-4377 or sales@onsetcomp.com.

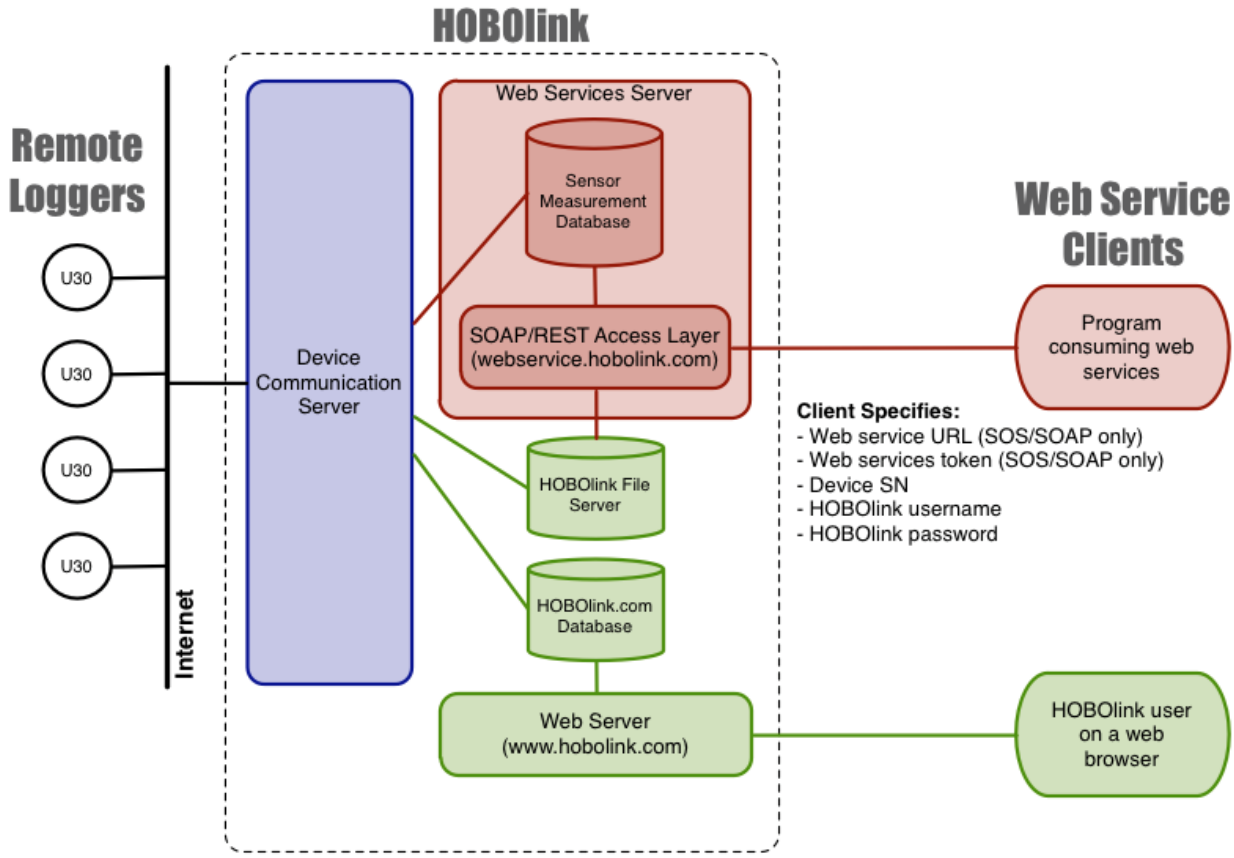
Requirements

- You should be a software engineer with a strong working knowledge of your programming language, its abilities, and its limitations.
- Ideally, you should have experience writing code that consumes SOAP or REST Web Services.
- You should have a HOBOlink account and a U30 Remote Monitoring Station, or plans to purchase one in the near future.

Diagram

The following diagram shows the interaction between HOBOlink Web Services, clients, and devices, including:

- The difference between a user hitting HOBOlink and a program hitting the web services.
- How the web services get data from the database, not directly from the devices.
- The parameters the web service program needs to specify.



Section 2: SOAP Web Services Tutorial

Step-by-Step Instructions

1. Get the Code

   The web service's methods are revealed through the WSDL file published by Onset at:

   https://webservice.hobolink.com/axis2/services/HOBOlinkAPIV2?wsdl

   The WSDL is an XML representation of the web service's interface. The same WSDL is used for our test and production environments.

   NOTE: The WSDL may contain other web service offerings. You should not attempt to call these other services.

   Most programming languages and/or IDEs have tools available to help with this conversion.

   Run the program appropriate to your environment that converts the WSDL to usable code. This will vary depending on your programming language.

   The previous API documented in earlier versions of this guide is available if needed. Contact Onset for more information.

Tips	
Java	• We recommend using axis2 to generate Java code from the WSDL: Java - http://axis.apache.org In particular, check out the documentation that explains the command line tool and available plug-ins: http://axis.apache.org/axis2/java/core/docs/reference.html http://axis.apache.org/axis2/java/core/tools/index.html • Let your IDE do the work for you. If there is a plug-in for your IDE listed on the axis2 tools page or in the documentation for your IDE, install, and run it. We have successfully used both NetBeans and IntelliJ to create test clients for our web services. • Local java classes will be created that represent the web service interface.
C#	• See: C# - http://msdn.microsoft.com/en-us/library/7h3ystb6.aspx
PHP4	• See : PHP4 - http://www.nusphere.com/php_script/nusoap.htm
PHP5	• See : PHP5 - http://sourceforge.net/projects/wsdl2php/
Ruby	• See : Ruby - http://www.tutorialspoint.com/ruby/ruby_web_services.htm

2. Contact Onset for Required Information

   Before you run client code, contact Onset Sales or Technical Support (1-800-LOGGERS) to obtain the necessary authentication and validation information. You will be provided with:

   - A device serial number and HOBOlink username/password, for use in testing
   - One generic validation token for use in testing
   - One individualized token you can use in your production environment with your purchased devices.

3. Write Client Code

   Referencing the web service interface generated in Step 1, write client code that calls the available methods. Your code needs to specify the web service URL, HOBOlink username, password, token, device serial number(s), and

sensor serial number(s). Other details such as time period and other filters are also available through the interface code. It is up to you whether you want to use a time period or other data filters when calling the web service.

The top-most method of the Sensor Observation Service is *getObservationFull*, which takes a query object as its argument. You will populate the query with the details listed above, before passing it to *getObservationFull.* The details of this and all the objects contained in this web service are defined in the Sensor Observation Service section in Section 3.

In your test code, use the test values provided by Onset along with the following endpoint URL:

https://webservice-dev.hobolink.com/axis2/services/HOBOlinkAPIV2

The test account should help you to get your basic framework up & running. You will not be able to run your code until you finish Step 4: Configure Client for SSL.

Once your tests are working properly, modify your client code to request data from your device serial number and sensors, using your own HOBOlink account information and your individualized validation token. Make sure to also remove the port (:89) from your web service URL, which will allow you to call into our production environment.

4. Configure Client for SSL

   Getting your client code to connect to an SSL URL will vary depending on your code. Generally speaking, you will need to add the certificate from the web service server to a trusted certificate file, and then reference that from within your code.

5. Write SensorML Parsing Code

   Data will be returned to the caller in SensorML format. If necessary, Onset can provide sample Java data binding code to marshal SensorML into Java objects.

Guidelines

- Bad or missing sensors will record -888.88 as their reading. You should screen for this value.
- The maximum number of the measurements that will be returned in a single call to the getObservationRequest service is 20,000.
- There is a small chance that duplicate entries can exist for the same sensor at the same time. You should ensure that your code can handle duplicate measurements.
- Observations are returned to the caller in UTC, not the local time of the device.
- The web services do not use the special XML markup CDATA to encode the SensorML in the SOAP response. Some programming languages may need to do additional processing to unescape this response. In our testing, the Java and C# languages needed no additional processing.

Section 3: Reference

Sensor Observation Service

- The WSDL can be found at https://webservice.hobolink.com/axis2/services/HOBOlinkAPIV2?wsdl
- The Sensor Observation Service endpoint is located at https://webservice.hobolink.com/axis2/services/HOBOlinkAPIV2
- getObservationFull (ObservationQueryFull query)
- Returns: String of Sensor Observation Service/O&M xml

Parameters

Hierarchy

The following illustration shows the relationship between parameters.

query **ObservationQueryFull**
- userName
- serialNumbers
- timePeriod **TemporalFilter**
  - operator
  - time **TimePeriod**
    - start
    - end
    - startIndet
    - endIndet
  - preset
- sensors **SimpleSensor**
  - sensorSerialNumber
  - measurementType
- filters **ComparisonFilter**
  - operator
  - sensor **SimpleSensor**
    - sensorSerialNumber
    - measurementType
    - value
    - value upper

auth **Authentication**
- token
- user
- password

Parameter Information

Name	Description
query	The object defining the query itself. Required: Y - Type: ObservationQueryFull
• userName	HOBOlink account user. If this parameter is populated, data will be returned for all devices that have been registered by that user. The serial_numbers parameter does not need to be populated. Either a list of serial number(s) or a user is required. Required: N - Type: String
• serialNumbers	Serial numbers of the U30 devices. If this parameter is populated, data will be returned for the specified devices. The user parameter does not need to be populated. Either a list of serial number(s) or a user is required. Required: N - Type: Array of String Objects
• timePeriod	Programming object that represents a time based filter of the data set. Required: N - Type: TemporalFilter
○ operator	The type of time based filter. Supported values are "Before", "After" and "During". Required: Y Type: String
○ time	Time Period for which the data is desired. Required: Y - Type: TimePeriod
▪ start	Start time in ISO 8601 format for the data to be returned. Required for the After and During operators. Required: N - Type: String
▪ end	End time in ISO 8601 format for the data to be returned. Required for the Before and During operators. Required: N - Type: String
▪ startIndet	Can be used in lieu of the start parameter to represent an indeterminate time. Supported value is "Now". Required: N - Type: String
▪ endIndet	Can be used in lieu of the end parameter to represent an indeterminate time. Supported value is "Now". Required: N - Type: String
○ preset	Use to preset the time rather than using a TimePeriod object. Valid arguments are: "Past_24", "Past_1", "Since_Midnight" Required: N - Type: String
• **sensors**	The list of sensors to be returned in the data set. Required: N - Type: Array of SimpleSensor objects
▪ sensorSerialNumber	Serial number of the sensor. Required: Y - Type: String
▪ measurementType	Measurement type of the sensor. Required: Y - Type: String
• **filters**	Programming object that represents a comparison based filter of the data set. Required: N - Type: Array of ComparisonFilter objects
○ ComparisonFilter	
▪ operator	The type of comparison based filter. Supported values are "PropertyIsEqualTo", "PropertyIsNotEqualTo", "PropertyIsLessThan", "PropertyIsGreaterThan", "PropertyIsLessThanOrEqualTo" or "PropertyIsGreaterThanOrEqualTo". Required: Y - Type: String
▪ sensor	Programming object that represents a sensor. Required: Y - Type: SimpleSensor

Name	Description
– sensorSerialNumber	Serial number of the sensor. Required: Y - Type: String
– measurementType	Measurement type of the sensor. Required: Y - Type: String
▪ value	The value to be filtered upon. Required: Y - Type: String
▪ value upper	Required: Y - Type: String
Authentication	Required: Y - Type: Object
• token	Authenticates the caller as a valid CDS Full consumer. Contact Onset Computer for a free token. SSL must be used to ensure encryption of this information. Required: Y - Type: String
• user	HOBOlink account under which the specified serial number is registered. Required: N - Type: String
• password	Password for the specified HOBOlink account. SSL must be used to ensure encryption of this information. Required: N - Type: String

Error Codes

Configuration

- CFG-001 – Error loading logging properties file.
- CFG-002 – Error loading the webservice properties file.

Database

- DBM-001 – Database error validating token.
- DBM-002 – Database error retrieving communication plans by VAR.
- DBM-003 – Database error retrieving HOBOlink status.
- DBM-004 – Database error retrieving the user.
- DBM-005 – No active deployments for user.
- DBM-006 – Database error retrieving active deployments for user.
- DBM-007 – Database error retrieving the device by serial number.
- DBM-008 – Database error retrieving the communications plan.
- DBM-009 – Database error saving the device.
- DBM-010 – Database error retrieving the deployment by device.
- DBM-011 – No valid communication plans for the respective VAR.
- DBM-012 – No subscribers found.
- DBM-013 – Database error retrieving list of subscribers.
- DBM-014 – Database error retrieving subscriber.

- DBM-015 – Database error retrieving credit type.
- DBM-016 – Database error saving subscriber.
- DBM-017 – Database error retrieving subscriber's webservice.
- DBM-018 – Database error retrieving webservice.
- DBM-019 – Database error saving subscriber's webservice.
- DBM-020 – No text files for deployment.
- DBM-021 – Database error retrieving latest text file for deployment.
- DBM-022 – Device is terminated.

Authentication

- ATH-001 – Invalid token.
- ATH-002 – Invalid user.
- ATH-003 – Encryption algorithm error during authentication.
- ATH-004 – Invalid password.
- ATH-005 – Invalid device serial number.
- ATH-006 – Invalid communications plan.
- ATH-007 – Invalid status.
- ATH-008 – Device not owned by respective user.

Email

- EML-001 – Error sending plan renewal confirmation email.

I/O

- IOE-001 – Could not read text file.
- IOE-002 – Error communicating with Sensor Observation Service.
- IOE-003 – The Sensor Observation Service reported an error.

Validation

- VAL-001 – Time period start time is null.
- VAL-002 – Time period start time is in the future.
- VAL-003 – No data found in specified time range.
- VAL-004 – Device serial number or user name is required.
- VAL-005 – Unsupported preset time period.
- VAL-006 – Unsupported indeterminate value for time period.
- VAL-007 – A start time is required for time period with the "during" operator.
- VAL-008 – An end time is required for time period with the "during" operator.
- VAL-009 – A start time is required for time period with the "after" operator.
- VAL-010 – An end time is required for time period with the "before" operator.
- VAL-011 – Invalid time period operator.

- VAL-012 – An operator is required with a time period.
- VAL-013 – Sensor information is required with "greater than" filter operator.
- VAL-014 – The value is required with "greater than" filter operator.
- VAL-015 – Sensor information is required with "less than" filter operator.
- VAL-016 – The value is required with the "less than" filter operator.
- VAL-017 – Sensor information is required with "greater than or equal to" filter operator.
- VAL-018 – The value is required with the "greater than or equal to" filter operator.
- VAL-019 – Sensor information is required with "less than or equal to" filter operator.
- VAL-020 – The value is required with the "less than or equal to" filter operator.
- VAL-021 – Sensor information is required with "equal to" filter operator.
- VAL-022 – The value is required with the "equal to" filter operator.
- VAL-023 – Sensor information is required with "not equal to" filter operator.
- VAL-024 – The value is required with the "not equal to" filter operator.
- VAL-025 – Invalid filter operator.
- VAL-026 – An operator is required with a filter.
- VAL-027 – Request is null.

Consuming Web Services

Java - http://axis.apache.org

C# - http://msdn.microsoft.com/en-us/library/7h3ystb6.aspx

PHP4 - http://www.nusphere.com/php_script/nusoap.htm

PHP5 - http://sourceforge.net/projects/wsdl2php/

Ruby - http://www.tutorialspoint.com/ruby/ruby_web_services.htm

SensorML

http://www.opengeospatial.org/standards/sensorml

XML Data Binding

Onset has successfully generated data binding code with the Java toolset XMLBeans (http://xmlbeans.apache.org/). The SensorML schema was too complex for basic Java tools like Castor or JAXB. To date, we have not investigated any XML data binding tools for other programming languages.

Section 4 : Sample SOAP Code

Java

```
import com.onset.aurora.webservices.GetObservationFullDocument;
import com.onset.aurora.webservices.GetObservationFullResponseDocument;
import com.onset.aurora.webservices.HOBOlinkAPIV2Stub;
import com.onset.aurora.webservices.beans.xsd.Authentication;
import com.onset.aurora.webservices.beans.xsd.ObservationQueryFull;
import com.onset.aurora.webservices.beans.xsd.TemporalFilter;
import com.onset.aurora.webservices.beans.xsd.TimePeriod;
import org.apache.axis2.AxisFault;

import java.net.URL;
import java.util.Calendar;

// This example uses the Axis2 WSDL2Code Maven plugin to generate Java client
// stubs from the HOBOlink Web Services WSDL file located at:
//
//     https://webservice.hobolink.com/axis2/services/HOBOlinkAPIV2?wsdl
//
public class GetObservationFullExample {

    // The cert file needs to contain the certificate at
https://webservice.hobolink.com
    final private static String CERT_FILE = "src/test/config/jssecacerts";

    // The service is defined by its endpoint:
    // dev:    https://webservice-dev.hobolink.com/axis2/services/HOBOlinkAPIV2
    // stable: https://webservice.hobolink.com/axis2/services/HOBOlinkAPIV2
    final private static String WS_URL = "https://webservice-
dev.hobolink.com/axis2/services/HOBOlinkAPIV2";";

    final private static String TOKEN = "1vHXNvx1QC";
    final private static String USER = "MRDTester";
    final private static String PASSWORD = "onset123";


    public static void main(String args[]) {
        // This line is required for SSL to work.
        System.setProperty("javax.net.ssl.trustStore", CERT_FILE);

        try {
            HOBOlinkAPIV2Stub stub = new HOBOlinkAPIV2Stub(WS_URL);

            // Set up the request with serial numbers, authentication object,
            // time period, and other filters
            GetObservationFullDocument reqDoc =
GetObservationFullDocument.Factory.newInstance();
            GetObservationFullDocument.GetObservationFull req =
reqDoc.addNewGetObservationFull();

            // Setup authentication parameter
            req.addNewAuth();
            req.getAuth().setToken(TOKEN);
            req.getAuth().setUser(USER);
            req.getAuth().setPassword(PASSWORD);

            // Setup query parameter
            req.addNewQuery();

            // Serial number filter
```

```
            req.getQuery().addSerialNumbers("1216485");

            // Time period filter
            req.getQuery().addNewTimePeriod();
            req.getQuery().getTimePeriod().setOperator("after");
            req.getQuery().getTimePeriod().addNewTime();
            req.getQuery().getTimePeriod().getTime().setStart("2013-12-
01T00:00:00+00:00");

            System.out.println("Request: " + reqDoc);
            // Execute web service
            GetObservationFullResponseDocument response =
stub.getObservationFull(reqDoc);
            System.out.println("Response: " +
response.getGetObservationFullResponse().getReturn());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

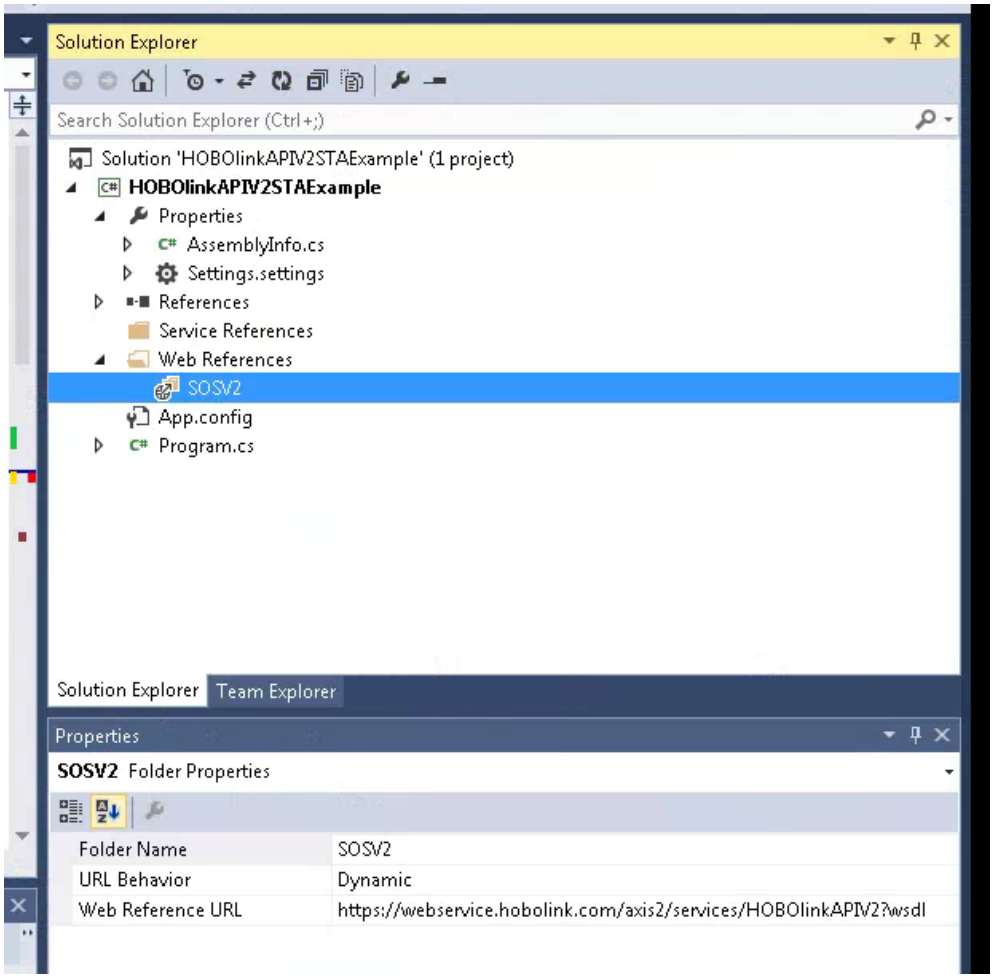
.NET/C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using HOBOlinkAPIV2STAExample.SOSV2;

namespace HOBOlinkAPIV2STAExample
{
    class Program
    {
        static void Main(string[] args)
        {
            HOBOlinkAPIV2 stub = new HOBOlinkAPIV2();

            Authentication auth = new Authentication();
            ObservationQueryFull query = new ObservationQueryFull();

            auth.token = "your token";
            auth.user = "your user";
            auth.password = "your password";

            string[] serialNumbers = { "your logger serial numbers" };
            query.serialNumbers = serialNumbers;

            query.timePeriod = new TemporalFilter();
            //query.timePeriod.preset = "Past_24";
            query.timePeriod.@operator = "During";
            //query.timePeriod.@operator = "After";

            query.timePeriod.time = new TimePeriod();
            query.timePeriod.time.start = "2014-12-10T13:00:00+00:00";
            query.timePeriod.time.end = "2014-12-10T14:00:00+00:00";

            query.sensors = new SimpleSensor[1];
            query.sensors[0] = new SimpleSensor();
            query.sensors[0].sensorSerialNumber = "logger SN:sensor SN";
```

```
            try
            {
                String result = stub.getObservationFull(auth, query);
                Console.WriteLine("The result is: " + result);
                Console.WriteLine("done");

                System.IO.File.WriteAllText(@"ws-response.xml", result);
            }
            catch (Exception e)
            {
                Console.WriteLine("Error: " + e);
            }
        }
    }
}
```

Example Project

Set up a Web Reference for the HOBOlink web service as shown below:



PHP4, PHP5, Ruby - Not available at this time

Section 5: Onset REST Web Services

Onset's REST Web Services allow you to easily obtain the latest data files from both public and private devices and incorporate them into your application. This section explains the URLs for obtaining the data files and provides an example application for using the REST Web Services.

Getting the latest data file for a device

If a device has been made public in HOBOlink (via HOBOlink Settings), anyone can access its data via the public URL at http://webservice.hobolink.com/rest/public/{some rest service}. Public services can be executed without authentication or the need to use SSL.

If a device has not been made public, its data can only be accessed via a private URL, which is http://webservice.hobolink.com/rest/private/{some rest service}. Private services require authentication and must be executed over SSL. The authentication scheme is Basic Authentication over SSL (secure sockets layer). An authenticated user can only access data for devices that the user has registered in his or her HOBOlink account, or have been made public.

The data file service public URL follows this convention:

http://webservice.hobolink.com/rest/public/devices/<serial_number>/data_files/latest/<data_file_type: dtf or txt>

and the data file service private URL follows this convention:

http://webservice.hobolink.com/rest/private/devices/<serial_number>/data_files/latest/<data_file_type: dtf or txt>

where <serial_number> is the device serial number, and <data_file_type: dtf or txt> is either dtf (to get the dtf file) or txt (to get the text file).

Example URLS:

Getting the latest .dtf file for a public device with serial number 123456:
http://webservice.hobolink.com/rest/public/devices/123456/data_files/latest/dtf

Getting the latest .txt file for a public device with serial number 123456:
http://webservice.hobolink.com/rest/public/devices/123456/data_files/latest/txt

Getting the latest .dtf file for a private device with serial number 123456:
https://webservice.hobolink.com/rest/private/devices/123456/data_files/latest/dtf

Getting the latest .txt file for a private device with serial number 123456:
https://webservice.hobolink.com/rest/private/devices/123456/data_files/latest/txt

Testing the Code

The URLS in the previous section point to a stable server with actual user devices. It is strongly recommended that you get your code running, tested, and debugged against the HOBOlink development environment before switching to a stable server. To use the development instance of our REST Web Services for testing, use this URL for public devices:

http://webservice.hobolink.com:88/rest/public/devices/1216485/data_files/latest/dtf

and this URL for private devices:

https://webservice.hobolink.com:89/rest/private/devices/1216485/data_files/latest/dtf

Example Application

This section describes an example application that demonstrates how to consume the REST Web Services using Java. If you plan to write your client in Java, you can use this example application as a starting point for consuming REST services for your own applications. If you plan to write your client in another language, refer to documentation provided by your API that describes how to consume REST Web Services in that language. Onset does not provide support for writing your own web service clients outside of what is contained in this document.

Overview

You can access Onset's REST Web Services using any language you choose, provided you have an understanding of how to access HTTP content in that language. While the example client provided here is written in Java, many other mainstream languages, such as Perl and .NET, have libraries you can use to consume RESTful web services.

This example uses a Java library called the Jersey Client API for RESTful Web Services. For information about the Jersey Client API, visit https://jersey.dev.java.net/ or search for "Consuming REST Web Services with Jersey" on the web. You may also use other Java libraries, such as Apache HttpClient at http://hc.apache.org/httpclient-3.x/.

To download the source code for this example application, go to:
https://webservice.hobolink.com/rest/info/ExampleRestWebServicesClient.zip

Or, view the complete source code at https://webservice.hobolink.com/rest/info/completeSource.html.

What the example application does

The example calls the private Onset REST Web Service to get the latest data files for a given device's serial number in two formats. The first file retrieved from the service in the example is a binary .dtf file and the second is in text format (.csv). After the example application retrieves the files, it saves them to the local file system. This example also demonstrates how to authenticate over SSL (secure sockets layer) to the private Onset REST Web Services.

Note: There are also public Onset REST Web Services that do not require authentication or SSL. To use these public services, simply skip the steps shown for SSL and authentication. The use of the Client to WebResource is the same. The only requirement for accessing a device's data from the public service is that the device has to have been made public. To make a device public, log onto HOBOlink, click the Settings tab, and check the appropriate box under Preferences > By Device.

Code walkthrough

The first section of code is the starting point or main method:

```
public static void main(String args[]){
   OnsetRestWebserviceExample client = new OnsetRestWebserviceExample();
   client.getLatestDataFile();
   System.out.println("success");
}
```

In this method, you create an instance of OnsetRestWebseviceExample and save it to an instance variable called client:

```
OnsetRestWebserviceExample client = new OnsetRestWebserviceExample();
```

Once the client is created, you can then call the public method "getLatestDataFile()" on it. If this is successful, "success" prints out to the console.

```
client.getLatestDataFile();
```

The getLatestDataFile() method is where all of the work is done, as explained below.

```
public void getLatestDataFile() throws UniformInterfaceException {
   //create default client config
   ClientConfig config = new DefaultClientConfig();
```

```
    // --setup the SSL configuration--
    //create a hostname verifier
    HostnameVerifier hostnameVerifier = getHostnameVerifier();
    //create the SSL context
    SSLContext sslContext = getSSLContext();
    //create and populate the https properties with the hostname verifier //and SSL
    context you created
    HTTPSProperties httpsProperties = new HTTPSProperties(hostnameVerifier,
    sslContext);
    config.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES,
    httpsProperties);

    //now that you have SSL configured, create a Client with the config you
    //created
    Client client = Client.create(config);

    //create a username and password
    String username="joe_user";
    String password="my_password";


    //now create an HTTP Basic Authentication filter that holds the //username and
    password
    HTTPBasicAuthFilter basicAuthenticationCredentials = new
    HTTPBasicAuthFilter(username, password);

    //pass the authentication credentials to the client object
    client.addFilter(basicAuthenticationCredentials);

    //create a web resource that points to the URL that you want
    WebResource webResource =
    client.resource("https://webservice.hobolink.com/rest/devices/<your U30
    SN>/data_files/latest/dtf");

    //Call "GET" on your web resource. You must provide the "GET" method //with the
    class type that you expect to get back. This could be //different class types
    depending on the service URL you are calling. //Some URLs may return a String
    for example.
    //For this particular example, we know that the service is passing back //a
    Java File type.
    File file = webResource.get(File.class);
    //done, process the file as needed.
    saveFileToLocalFilesystem(file,
    "/Users/gmirabito/webServiceTest/dtf_file_from_rest_service.dtf");

    //change the a web resource to get a text file
    webResource =
    client.resource("https://webservice.hobolink.com/rest/private/devices/<your
    U30 SN>/data_files/latest/txt");

    //This time, since we are requesting text data, we will get the data as //a
    string. We could have used a file, but we are demonstrating the
    //automatic type conversion of the library.
    String textFileData = webResource.get(String.class);
    System.out.println("text file data = " + textFileData);

    //Now we will make the same web service call, but instead if getting a //string
    back, we will get the result as a file.
    //The only change here is that we pass the "GET" method a File.class //instead
    of a String.class
    File textFile = webResource.get(File.class);
    saveFileToLocalFilesystem(textFile,"/Users/gmirabito/webServiceTest/text_file_f
    rom_rest_service.txt");

}
```

Create Client Configuration Object

Looking at the code in more detail, you would create a ClientConfig object to hold the security configuration. The client configuration needs the following items to perform encryption and authentication:

- HostnameVerifier: This allows you to verify a hostname for the webservice.
- SSLContext: This allows you the verify SSL information in your program, allowing you to accept or reject the SSL response.
- HttpProperties: This object simply holds the HostnameVerifier and SSLContext you created.

Create the HostnameVerifier by calling the method getHostnameVerifier. This object implements one method called verify(). This method returns a boolean (true or false). It allows your code a chance to verify the hostname of the server. If you return "true," the hostname is considered OK and the web service call can continue. If it returns "false," the hostname is not trusted and the webservice call will abort. This example returns "true."

Next, set up the SSLContext by calling getSSLContext() on the object. The SSLContext allows you to inspect the SSL information to decide if you should trust it. This example accepts all SSL certificates. Implement your own logic to decide whether to trust a certificate. The SSLContext requires you to implement a javax.net.ssl.TrustManager. This example uses a javax.net.ssl.X509TrustManager. The TrustManager must implement the following methods:

```
public void checkClientTrusted(java.security.cert.X509Certificate[] arg0, String
arg1) throws java.security.cert.CertificateException {
public void checkServerTrusted(java.security.cert.X509Certificate[] arg0, String
arg1) throws java.security.cert.CertificateException {
public java.security.cert.X509Certificate[] getAcceptedIssuers() {
```

You can use these methods to decide whether to trust the SSL certificate. The example application trusts everything.

Create an instance of HTTPProperties, passing in the hostnameVerifier and sslContext.

```
HTTPSProperties httpsProperties = new HTTPSProperties(hostnameVerifier,
sslContext);
```

After the HTTPProperties object is created and populated with the security information, put them into the ClientConfig by calling:

```
config.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES,
httpsProperties);
```

Summary:

```
//create default client config
ClientConfig config = new DefaultClientConfig();

// --setup the SSL configuration--
//create a hostname verifier
HostnameVerifier hostnameVerifier = getHostnameVerifier();
//create the ssl context
SSLContext sslContext = getSSLContext();
//create and populate the https properties with the hostname verifier and //SSL
context you created
HTTPSProperties httpsProperties = new HTTPSProperties(hostnameVerifier,
sslContext);

config.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES,
httpsProperties);
```

Create Client Object

Use the ClientConfig object to create a client object.

```
Client client = Client.create(config);
```

Next, create the authentication credentials by creating an HTTPBasicAuthFilter and passing it the username and password.

```
HTTPBasicAuthFilter basicAuthenticationCredentials = new
HTTPBasicAuthFilter(username, password);
```

This filter is used to provide the username and password to log in to the webservice. Add HTTPBasicAuthFilter to the client object with:

```
client.addFilter(basicAuthenticationCredentials);
```

Summary:

```
//create a username and password
String username="joe_user";
String password="my_password";
//now create an HTTP Basic Authentication filter that holds the username and
password
HTTPBasicAuthFilter basicAuthenticationCredentials = new
HTTPBasicAuthFilter(username, password);

//pass the authentication credentials to the client object
client.addFilter(basicAuthenticationCredentials);
```

Call the Service

To call the service, start by getting an instance of WebResource from the client. This is done by calling:

```
WebResource webResource =
client.resource("https://webservice.hobolink.com/rest/devices/<your U30
SN>/data_files/latest/dtf");
```

Notice that the actual URL of the service we want to call is passed. **Note:** Since the client is configured with the SSL information as well as the username and password, authentication is handled automatically.

With the WebResource instance you can call "get" on it to get the data you need. The example application gets the latest data file from the service. A benefit of the Jersey client library is that it can handle web content in a dynamic way that allows you to get data from the web in the form of Java objects you can use. For example, because you are getting the latest data file, the easiest way to work with the data returned from the service is to use it as a Java File object. To do this, pass in the type of object you want back from the service as follows:

```
File file = webResource.get(File.class);
```

Notice that the web resource's "get" method is called with a Java File.class. This means the context returned from the service will be passed back as a Java File object. Once you have this file, you can use it as you wish. The example application calls the saveFileToLocalFilesystem method. This method takes in the file received from the web service and the path on the filesystem where it should be saved. In this case, the file is saved to /Users/joeuser/webServiceTest/dtf_file_from_rest_service.dtf.

```
saveFileToLocalFilesystem(file,
"/Users/joeuser/webServiceTest/dtf_file_from_rest_service.dtf");
```

Use the same client to make another webservice call to get a text file for the same device. Instead of getting back a Java File, ask for a Java String in return and then print it out to the console.

```
//change the a web resource to get a text file
webResource =
client.resource("https://webservice.hobolink.com/rest/private/devices/<your U30
SN>/data_files/latest/txt");

//This time, since we are requesting text data, we will get the data as a
//string. We could have used a file, but we are demonstrating the
//automatic type conversion of the library.
String textFileData = webResource.get(String.class);
System.out.println("text file data = " + textFileData);
```

Finally, call the save service and ask for the text file as a Java file instead of a String object. Once you have the text file, save it to the filesystem location at /Users/joeuser/webServiceTest/text_file_from_rest_service.txt.

```
//Now we will make the same web service call, but instead if getting a string
back, we will get the result as a file.
//The only change here is that we pass the "GET" method a File.class instead of a
String.class
File textFile = webResource.get(File.class);
saveFileToLocalFilesystem(textFile,"/Users/joeuser/webServiceTest/text_file_from_r
est_service.txt");
```

Note: If you want to use the public REST web services, you do not need to set up a client configuration. You can simply create a client, call client.resource and then call "get" on the web resource.

Running the Example Java Application

Requirements:

- Maven2 to build the project. Check the version of Maven installed by typing **mvn -version** on the command line. If you don't already have maven installed, visit http://maven.apache.org.
- Java 1.5 or later. Check the version of Java installed by typing **java –version** on the command line.

Build the source

1. Download the source at https://webservice.hobolink.com/rest/info/ExampleRestWebServicesClient.zip
2. Extract the zip file to any location.
3. Open the file src/main/java/com/onset/test/rest/webservices/OnsetRestWebServiceExample.java.
4. Within the source file, edit the username and password to something valid.
5. In the source file, change the web service URLs to include the serial number for your device.
6. In the same file, change the file paths on the calls to *saveFileToLocalFilesystem(textFile,"/Users/gmirabito/webServiceTest/text_file_from_rest_service.txt")* to a valid file path on your system.
7. On the command line, navigate to the directory where you extracted the source and run the following Maven command:
   mvn package
   This will package the application in an executable jar with dependencies. The executable jar can be found in the "target" directory with the name OnsetRestWebServiceExample-jar-with-dependencies.jar.

Run the application

To run the example, execute this command from the target directory of the project you just built:

```
java -jar OnsetRestWebServiceExample-jar-with-dependencies.jar
```

You should see "success" printed out on the console.

Section 6: Glossary

SOAP	Simple Object Access Protocol. A protocol for exchanging XML-based messages over computer networks.
REST	Representative State Transfer. An architectural style that allows for point-to-point communication over HTTP using XML.
WSDL	Web Services Description Language. Describes the web service interface. Typically used to develop web service clients.
Stub	Also known as skeleton code. A method of generating class files based off a web service definition file (WSDL). Most programming languages have SOAP tools designed to generate stub code from a WSDL file (e.g. wsdl2java).
Sensor Observation Service	Provides an API for managing deployed sensors and retrieving sensor data and specifically "observation" data.
Endpoint	A web service endpoint is a (referenceable) entity, processor, or resource to which web service messages can be addressed.
Marshal	The act of converting an XML document to and from a programming object.
Bind	The process of representing the information in an XML document as an object in computer memory.