

---

# WeatherNet: The Observers Network

*Release 0.2.1*

Christopher Blunck

March 27, 2003

[blunck@gst.com](mailto:blunck@gst.com)

## ABSTRACT

WxNet is a network of meteorologists, publishing and consuming data according to a well defined interface described in WSDL *The Web Services Description Language 1.1*. WeatherNet allows any weather observer to share and gather information over a commonly understood network using industry standard protocols. It also provides a mechanism by which an observer can register with the network and search the network for other observers based upon their geographic location.

WxNet is platform independant, but the reference implementations require Python 2.0 or later and ZSI obtained from CVS.

The WxNet homepage is at <http://meta-tools.sf.net/>.

## COPYRIGHT

This work is published under the GNU General Public License.  
All Rights Reserved.

## Acknowledgements

I am extremely grateful to Rich Salz of the ZSI project <http://pywebsvcs.sf.net/> for his enourmous contribution to the Python for Web Services community. Without his code, WeatherNet would not be possible. Thanks, Rich.

I am also grateful to Neal Norwitz and Eric Newton of <http://www.metashash.com/> for introducing me to Python and for answering my barrage of annoying questions. Thanks guys.



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Preface . . . . .	1
1.2	Intended Audience . . . . .	1
1.3	Architectural Background . . . . .	1
1.4	WeatherNet Architecture . . . . .	2
<b>2</b>	<b>The WeatherNet Interface</b>	<b>3</b>
2.1	Complex Data Types . . . . .	3
2.2	Operations . . . . .	4
<b>3</b>	<b>WeatherNet Servers</b>	<b>7</b>
3.1	Reference Implementation . . . . .	7
3.2	Serialization . . . . .	8
<b>4</b>	<b>WeatherNet Clients</b>	<b>9</b>
4.1	Reference Implementation . . . . .	9
4.2	Deserialization . . . . .	10
<b>5</b>	<b>WeatherNet Registry</b>	<b>11</b>
5.1	Registering . . . . .	11
5.2	Unregistering . . . . .	11
5.3	Listing Nodes . . . . .	12
5.4	More coming... . . . .	12
<b>6</b>	<b>Summary</b>	<b>13</b>



# Introduction

WxNet, WeatherNet, is a network of weather observers who share and discover meteorological data using well defined APIs over a standard network interface. When connected to the network, any WeatherNet-aware client can connect and receive meteorological data from any station on the network. This data could be used for graphical display, forecasting, or any other general purpose. Similarly, weather providers are able to connect to the network and share data with anyone they like.

WxNet is platform independant, but the reference implementation requires Python 2.0 or later as well as ZSI available from `:pserver:anonymous@cvs.sourceforge.net/cvsroot/pywebsvcs`.

The WxNet homepage is at <http://meta-tools.sf.net/wxnet>.

## 1.1 Preface

The reference implementations for WeatherNet are written in Python. As a result, all class definitions and sample code are in python. Because WeatherNet is an interface definition, any language can be used to implement clients or servers - do not be duped into thinking that WeatherNet is python specific.

## 1.2 Intended Audience

WeatherNet represents a collective of weather observers, all of which share their meteorological data with the larger community. To harness the network, you must develop a client that conforms to the standards defined by WeatherNet. This document outlines those standards, and provides examples that can be used to develop a WeatherNet client or server. As such, programmers are the intended to be the primary audience of this document, but project managers and decision makers should still be able to glean some understanding from the knowledge contained in this document.

## 1.3 Architectural Background

This project centers around Web Services. The purpose of web services is to provide a standard format for defining publicly accessible operations implemented on a remote computer. Using web services, a service provider can define operations they have implemented as well as the data structures passed and returned from said operations. Although it is not required, most web services are accessed via *The SOAP 1.1 Specification*, which allows for transport independence. This means that a client can interact with a SOAP server via http, smtp, ftp, or any other protocol so long as it is explicitly stated in the definition of the interface. As a best practice, most people use http as a transport protocol for SOAP.

### 1.3.1 Simple Object Access Protocol

SOAP only defines the rules for binding to a remote operation. It provides serialization and deserialization of objects to XML (the language used by SOAP), but does not define the interface for a remote service. Simply put, SOAP defines a mechanism by which a client can serialize parameters to an operation, send them over a transport protocol, and deserialize the response. SOAP is implemented in nearly every language, and is a cornerstone of Microsoft's .NET initiative as well as other interoperability efforts.

### 1.3.2 Web Service Description Language

Web Service Description Language (hereafter referred to as WSDL) operates at a higher level and allows you to define services accessible via SOAP. Contained within a WSDL definition you will find a set of services, each of which containing multiple operations. For each operation, an input and output message is defined, which may (or may not) contain primitive and complex data structures. Complex data structures are defined in a schema section, and can contain a mixture of primitive and complex types. All the above concepts come together to form a complete description of a service (described in the language of WSDL) that is universally understandable.

## 1.4 WeatherNet Architecture

The core of the `WxNet` project surrounds the WSDL definition of operations a weather observer can implement. With the definition formalized, universal clients can be developed to access information on a much wider (and standard) scale than what is currently provided. Several other projects exist related to meteorological data sharing, but WeatherNet is the first to define standards (in WSDL) against which any client can be written.

A benefit of using WSDL is the decomposition that occurs when analysing a weather network service oriented architecture. In order to participate on the WeatherNet, an observer need only conform to the interface defined by this project. How you implement that interface is completely up to you: you could report meteorological data in real time, or you could pull said data from a database. It simply doesn't matter, so long as you implement the published interface.

Using the WeatherNet architecture, a variety of clients can be written. A reference implementation exists that connects to an observer on the WeatherNet and displays the current weather information. In the future, more complex clients could be written to make use of a single capability: collection of data from different geographic locations. The harvesting capability WeatherNet provides supports a variety of use cases, the most significant of which is more data to provide to forecasting models.

---

# The WeatherNet Interface

The heart of the WeatherNet project is the published WeatherNet interface, which all data providers participating on the network must implement. The definition of the interface is via the Web Services Description Language, providing a high level of interoperability between servers and clients.

## 2.1 Complex Data Types

WeatherNet defines three complex data types: `RainSummary`, `WindSummary`, and `Summary`, each of which corresponds to a python class. `Summary` is an aggregation of `RainSummary` and `WindSummary` as well as some additional fields that summarize the current state of a weather station. These includes vital information such as temperature, barometric pressure, dew point, sunrise (and sunset), as well as local station time.

The class definition for a `RainSummary` follows:

```
class RainSummary:
    def __init__(self, rate=0.0,
                 storm=0.0,
                 day=0.0,
                 month=0.0,
                 year=0.0,
                 stormStartDate=None):
        self.rate = rate
        self.stormTotal = storm
        self.dayTotal = day
        self.monthTotal = month
        self.yearTotal = year
        self.stormStartDate = stormStartDate
```

The fields contained within a `RainSummary` represent rain totals over various time periods. The `dayTotal`, `monthTotal`, and `yearTotal` fields represent the total rain accumulation (in inches) over the previous day, month, and year respectively. The values are not relative, meaning if it is February 15, the value in the `monthTotal` represents the total rain accumulation between February 1 and February 15, inclusive. Likewise, the day total is since midnight of the current day, as opposed to the total over the past 24 hours.

The class definition for a `WindSummary` follows:

```

class WindSummary:
    def __init__(self, speed=0, dir=0, speed10min=0):
        self.speed = speed
        self.direction = dir
        self.speed10min = speed10min

```

The fields contained within a `WindSummary` represent the current wind status, including the 10 minute historical average for the wind speed. The 10 minute historical wind speed average may or may not be of professional significance, but has been included because my person weather station reports it and I thought it best to share it just in case. The speed is an integer and is in mph, and the direction is also an integer representing the direction the wind is currently blowing from ([0 - 360]).

The class information for a `Summary` follows:

```

class Summary:
    def __init__(self, datestamp=None,
                 pressure=0.0,
                 temperature=0.0,
                 windsummary=None,
                 rainsummary=None,
                 sunrise=None,
                 sunset=None,
                 dewpoint=0.0,
                 humidity=0.0):
        self.datestamp = datestamp
        self.pressure = pressure
        self.temperature = temperature
        self.windsummary = windsummary
        self.rainsummary = rainsummary
        self.sunrise = sunrise
        self.sunset = sunset
        self.dewpoint = dewpoint
        self.humidity = humidity

```

A `Summary` represents a snapshot of the current meteorological data from an observer. Contained within are the complex types of `WindSummary` and `RainSummary`, both of which are described above. Additionally, a `Summary` defines important primitives like the current temperature (in Fahrenheit), barometric pressure (in inches of mercury), sunrise and sunset (standard python time tuples), dewpoint (in Fahrenheit), and the relative humidity. The `datestamp` field represents the local time of the station in Python time tuple format. A `Summary` is a good complex type to use as a client because it contains all information a provider on the WeatherNet can supply, and is easily transported.

## 2.2 Operations

WeatherNet currently operates in “read-only” mode, meaning weather providers are not required to service any interactive requests. The operations defined by WeatherNet are pull operations including: `getTemperature`, `getPressure`, `getWindSummary`, etc. In the future, the WeatherNet interface may be expanded to include interactive invocations, with use cases of the form: `getWindSummary(startDate, stopDate)` or `getTemperatureRange(startDate, stopDate)`. Since the community at this point is tiny, I have chosen not to define the interactive operations until such time that the community asks for it, or providers offer such operations.

The list of operations follows:

```

# wind sensors
def getWindSpeed()
def getWindDir()
def getWindSummary()

# rain sensors
def isCurrentlyRaining()
def getRainRate()
def getRainDay()
def getRainMonth()
def getRainYear()
def getStormStartDate()

# core sensors
def getTemperature()
def getPressure()
def getRelativeHumidity()
def getSunrise()
def getSunset()
def getDewPoint()
def getLocalTime()

# summary information
def getSummary()

```

### 2.2.1 Wind Sensors

WeatherNet requires the observer to report the current wind speed and direction. Both fields are integer values, the direction in the format of degrees. The 10 minute wind speed average is available in the WindSummary, but is not required.

### 2.2.2 Rain Sensors

WeatherNet requires the observer to report the current rain rate as well as the accumulation totals for the day, month, and year. Additionally, an observer should report the date in which the rain event started (if we are in a rain event), or None if the station is not currently in a rain event. The `isCurrentRaining()` method can be used to determine if it is currently raining at the station.

### 2.2.3 Core Sensors

WeatherNet requires the observer to report the current temperature (in Fahrenheit), barometric pressure (in inches of mercury), and relative humidity. The `getSunrise()` and `getSunset()` methods must be defined, but are not required to be implemented. It is sufficient to stub those methods as returning None.

The `getLocalTime()` operation must be implemented.



# WeatherNet Servers

Servers are producers of meteorological observations distributed via the WeatherNet network. The implementation details of a WeatherNet server are insignificant as long as the published interface is supported.

Additionally, a server programmer may choose to implement a superset or subset of the operations defined in WeatherNet to return actual data, but is not required to do so. In the situation where a server cannot (or does not wish to) provide complete coverage of the WeatherNet interface, unsupported operations should be stubbed in order to insure compliance with the standard. Thus, if a participant wishes to be a provider, but does not have an anemometer, they can simply return a WindSummary with speed, direction, and 10 minute average as 0, 0, and 0.

There is no support available (yet) for publishing of supersets of operations that WeatherNet does not provide. I have considered this, and believe that it will be an important problem to solve. But given the current lack of a community, I decided not to implement that functionality until requested.

## 3.1 Reference Implementation

Python was chosen as the language used for the reference implementation because the loosely typed nature of the language lends itself nicely to web services development. The libraries in Python (ZSI in particular) are extremely easy to use, and do not require the use of external tools (WSDL -i, Stub generators), making prototyping and development extremely painless and simple. This is a credit to both the language and the ZSI project for providing such an intuitive interface.

The server side reference implementation retrieves data from a database. This could easily be adapted to read data in realtime from a weather station, but given the native hardware issues involved (and the lack of support from our weather hardware supplier), this was both impractical and difficult. A database seemed to be a fairly platform independant location with an easy to understand interface, and thus it was selected as the platform for our reference implementation.

The code for the server side reference implementation is stored in the `server` module in `server/python` directory. As such we will not include the entire source code in this document. Full examination of the reference implementation is recommended, and this document is really intended to whet your appetite for a feature-rich example (the reference implementation contained with the WeatherNet distribution).

Contained below is a snippet of our server side implementation:

```

#!/usr/bin/env python

import sys, MySQLdb, time

from ZSI import *
from ComplexTypes import *

DB_HOST='localhost'
DB_USER='Home_Weather'
DB_PASSWD='Home_Weather'
DB='Home_Weather'

def getPressure():
    cursor.execute('select bar from Current')
    return cursor.fetchone()[0]

def getWindSummary():
    cursor.execute('select windspeed, winddir, windspeed10min from Current')
    speed, dir, speed10min = cursor.fetchone()
    return WindSummary(speed, dir, speed10min)

# bind to the database, and obtain a cursor
db = MySQLdb.connect(host=DB_HOST, user=DB_USER, passwd=DB_PASSWD, db=DB)
cursor = db.cursor()

# launch the service as a CGI, using wxnet as a namespace for complex types
if __name__ == '__main__':
    nsdict = { 'wxnet' : 'http://wxnet.dnsalias.com/WxNet' }
    dispatch.AsCGI(nsdict=nsdict)

```

This server side implementation demonstrates two functionalities: `getPressure` and `getWindSummary`. The `getPressure` function returns a primitive, and the `getWindSummary` function returns a complex type. As a provider, you may choose to report data in realtime, or you can (as we did) report data stored in a database. This is the benefit of coding to an interface - you have the ability to implement according to your own requirements, desires, capabilities, or restrictions.

Thankfully, ZSI supports complex structures. This allows you as a WeatherNet provider to return complex data structures (like `WindSummary`). For more information on the details of serialization and deserialization of complex structures in ZSI, see <http://pywebsvcs.sf.net/>.

## 3.2 Serialization

Because we are using ZSI, it is important to point out some behaviors that are required on the server side in order to work with the ZSI module. As a server, you must define a `TypeCode` to use for any complex data structures (`WindSummary`, `RainSummary`, and `Summary`), which allows ZSI to convert from a Python object to XML format suitable for SOAP transport. This is described in more detail in the client section on Deserialization.

# WeatherNet Clients

Clients are consumers of meteorological observations published via the WeatherNet network. Similar to a server, what a client chooses to do with the information is beyond the scope of the WeatherNet project. Because the WeatherNet interface is published (and thus adhered to by all server participants), a client has the ability to “roam” on the WeatherNet network, connecting to any server and displaying data in realtime (as long as the server provides data in realtime).

In the future, clients will be able to query a central node (the WeatherNet controller) to ask for a list of connected servers in the network. This list will most likely include the geographic location of each server, giving the client maximum flexibility in defining which server should be used.

## 4.1 Reference Implementation

The client side reference implementation uses Python for many of the same reasons that the server side reference implementation uses Python. Loosely typed classes work very well with web services, making it very easy to develop a web service proxy.

The goal of the client side interface is to connect to a server and ask for the current weather conditions. ZSI was used on the client side to provide connectivity (primarily because it supports both WSDL parsing and SOAP binding), but any language and any project (or COTS software) could be used in replacement of ZSI because WeatherNet operates at the WSDL level, and thus is a highly interoperable network.

Below is code that connects to a WeatherNet server, and requests some weather information:

```

#!/usr/bin/env python

from ZSI.wsdl import ServiceProxy
from ZSI import *
from ZSI.client import *

def main():
    wxnet = ServiceProxy('`http://wxnet.dnsalias.com/Weather/WxNet.wsdl`')
    pressure = wxnet.getPressure()

    print 'The current weather conditions are as follows:'
    print 'Pressure:    %.4f' % summary.pressure

    (speed, dir, speed10min) = wxnet.getWindSummary()
    print 'Wind:        %d at %d mph' % (speed, dir)

if __name__ == '__main__':
    sys.path.append('../..../common/python')
    main()

```

## 4.2 Deserialization

Because we are using ZSI, it is important to point out some behaviors that are required on the client in order to work with the ZSI module. As a client, you must use a module named `ComplexTypes` in order for transparent deserialization to work. In this example, the `WindSummary` is defined in the `ComplexType` module to enable the ZSI framework. It is defined below:

```

from ZSI import TC
class WindSummary:
    def __init__(self, speed=0, dir=0, speed10min=0):
        self.speed = speed
        self.direction = dir
        self.speed10min = speed10min
WindSummary.typecode = TC.Struct(WindSummary,
                                  [TC.Ilong('speed'),
                                   TC.Ilong('direction'),
                                   TC.Ilong('speed10min')],
                                  'wxnet:WindSummary')

```

If you do not store the definition for `WindSummary` in `ComplexTypes` and append it to `sys.path` prior to invoking your webservice, you will need to call `wxnet.ReceiveRaw(WindSummary)` after invoking the remote operations. As a best practice, WeatherNet plans to define a set of common libraries for various languages to facilitate information passing between client and server. Python is currently the only support language of such middleware classes.

A client is free to use the WeatherNet information in any way, shape, or form. It can be returned to the end user in graphical format, forwarded to another weather network or organization, or even fed as input into forecasting models for the National Weather Service.

---

# WeatherNet Registry

In order for WeatherNet servers to be discovered by clients, they must be registered in the WeatherNet Registry. Like the server interface, the registry interface is defined using WSDL. The two operations currently implemented are: `online(server)`, and `list()`.

## 5.1 Registering

To join the WeatherNet network and share meteorological data, you must invoke the `online` operation. The `online` operation accepts the following parameter:

```
class WeatherNetServer:
    def __init__(self, WSDL=None,
                 latitude=0.0,
                 longitude=0.0,
                 city=None,
                 state=None,
                 country=None,
                 operator=None):
        self.WSDL = WSDL
        self.latitude = latitude
        self.longitude = longitude
        self.city = city
        self.state = state
        self.country = country
        self.operator = operator
```

The item of key importance in the `WeatherNetServer` class is the `WSDL` attribute. It must be defined and accessible to any potential WeatherNet client. This means that it should not be behind a firewall, or network restricted in any other manner. In the future, the WeatherNet registry will routinely ping your node to verify connectivity.

When your node comes online, you should call the `online` method to announce to the rest of the network that you are open for business and ready to service requests.

## 5.2 Unregistering

If you disconnect from the network, your node will automatically be detected by the registry as being down. It will no longer be reported as a server returned from the `list()` operation.

## 5.3 Listing Nodes

All clients are going to want to contact the registry and request a list of connected nodes. This allows the client to present the list (perhaps graphically based upon the geographic information embedded within the return from `list()`) to the end user and allow them to select which node in the network to connect to.

The call to `list()` returns an array of `WeatherNetServer`, as defined above.

## 5.4 More coming...

In the future, the registry will become much more full-featured, giving network-wide information (number of connected servers and clients, featured sites, etc). Expect frequent additions to the registry, but for the interfaces already defined to be fairly static.

## Summary

WeatherNet is simply an interface that describes operations common to most weather observing stations. Because it is an interface definition, it allows for numerous client and server side applications, that can ultimately support a wide variety of business needs. WeatherNet uses industry standards such as WSDL and SOAP to facilitate interoperability, and aims to be as unobtrusive as possible.

It is hoped that the WeatherNet project will be used as a starting point for more complex meteorological network used by hundreds of amateur and profession meteorologists to publish and share data.