Build an app to read sensor data and predict failure using Informix TimeSeries service

Michael Chris Golledge (https://www.ibm.com/developerworks/community/profiles/html/profileView.do?key=f61406d7-1522-4263-b6aa-f8808c046327&tabid=dwAboutMe)
Advisory Software Engineer
IBM

10 July 2014

Preventing problems costs less than reacting to them. Prevention implies some knowledge of where things are going, and predicting where things will be in the future requires some knowledge of where things have been in the past. This is where IBM® Informix® TimeSeries comes into play. Not only does it provide a means of recording what has happened, it is easy to use to predict what might happen. In an internet-of-things context, any sensor can record its information into a time series data set. This demonstration is a simple simulation of a refrigerator, cloud-based storage and analytics, and monitoring software to alert the user if the device is approaching unsafe temperature limits.

Preventing problems costs less than reacting to them. But to prevent a problem, you must be able to predict a future issue based on knowledge of the past. Sensors often pick up signs of impending failure before it happens. Because sensors record information into a time series data set, you can use IBM Informix® TimeSeries to help you predict and prevent problems. Informix TimeSeries records changes over time in a format that is outside of the traditional relational table format. This format supports more efficient use of storage space and faster query processing.

IBM Bluemix provides an easy-to-use platform for developing a solution involving sensors, database storage (in the TimeSeriesDatabase), server-side analytics, and a monitoring application to warn the customer if trouble appears to be imminent. This demonstration is a simple simulation of a refrigerator, cloud-based storage and analytics, and monitoring software to alert the user if the device is approaching unsafe temperature limits. Use the tools described here to build a similar solution to solve a real problem you're facing.

> *“ This demo provides simple building blocks showing how to use a smart sensor to detect problems, in time to prevent them. ”*

In this demo, the sensor is a simulation of a thermometer on a fridge. The user moves the thermometer, the sensor sends its readings to the service in the cloud, and the monitor polls the recent statistics from the service. If the temperature of the fridge is likely to exceed safe limits in the near future, the monitor alerts the user. In a real application, the monitoring software would probably be run by a company servicing many devices, and the monitoring software would be looking for problems on all those devices.

What you'll need for your application

- General proficiency with Java™ programming, applets, and a Java development environment.
- The JDBC driver for IBM Informix.
- Moderate knowledge of the Informix TimeSeries feature.
- Access to two libraries from Apache:
  - Apache Common Lang, math library
  - Apache Wink

Run the app
Get the code at JazzHub

The components

The components of this application running outside of the cloud operating environment are Java applets. It is common for browsers to disable the running of applets by default. If you have trouble getting the applets to run, try enabling Java in your browser and enabling Java in your control panel.

The application contains three main modules: server-side classes, the sensor, and the monitor. The server-side classes run as servlets within the Liberty for Java runtime. The sensor and the monitor are separate, lightweight, Java applets that run in a web browser. The following tables give an overview of the classes and what they do.

Server-side class	Brief description
`TSConnection`	Handles all the direct database I/O.
`InitializeDB`	Routes request to create database objects to TSConnection. Should only be needed when modifying the application.
`ServiceStatus`	Only verifies that the service is online and sees what the VCAP_SERVICES entry is.
`SensorID`	Receives the request to register the sensor on the database and returns a unique ID to the sensor. An alternate model would be for the sensor to tell the server what its ID is.
`TSRead`	Routes requests from the monitor for recent sensor statistics to TSConnection.

`TSWrite`	Routes sensor readings to TSConnection.
`TSStats`	Encapsulates recent statistics and defines thresholds.
`TSType`	Encapsulates the time series row type.

Sensor class	Brief description
Sensor	Applet with graphical interface that transmits readings to the server.
`TSType`	Serializable type that encapsulates the data to be transmitted.

Monitor class	Brief description
Monitor	Applet with graphical interface that reads data analysis and signals user.
`TSStats`	Serializable type that encapsulates the data to be transmitted.

Let's go over some of the more interesting sections of code.

TSConnection.initializeDB method

Do not expose this function in a production environment. It is accessible in this demo so that you can see some of the feature content of the TimeSeries database service.

The `TSConnection.initializeDB` method limits the amount of data stored and defines a more granular calendar pattern when it is called.

Although you can use the predefined objects common to every instance of the TimeSeries service, this demo takes readings every second and limits how much data is kept. For that purpose, the demo requires some TimeSeries object types that do not come with the standard database. The following listings are for these objects.

Define the calendar pattern

Calendar definitions, as shown in the following code listing, do not matter as much with irregular series as they do with regular series, but I like to keep the object definitions consistent with their intended use. If I want to use a regular series at some point in the future, defining it now makes that easy. The shortest period that comes with the default database ready to use is 15 minutes.

```
// This statement is a little bit of a trick to create a new pattern
// if one does not already exist, and do nothing if it does. It saves
// having to code multiple statements and logic.
ignoreErr[i] = false;
statements[i++] = "insert into CalendarPatterns "
  + "select "
  +"    'onesec','{1 on}, second' "
  + "from sysmaster:sysdual "
  + "where not exists (select 1 "
  + "    from CalendarPatterns where cp_name = 'onesec')";
```

Establish time zone consistency

The simple row type has just a couple of fields.

```
statements[i++] = "create row type thermometer_reading ( "
 + "tstamp_gmt       datetime year to fraction(5), "
 + "celsius     real "
 + ")";
```

What is the time zone of the cloud? To eliminate ambiguity about what the values in the database mean, give your columns names that give meaning to the values stored there. For example, the `_gmt` suffix in the previous listing indicates that values are in Greenwich Mean Time. The actual time zone is enforced by the code that assigns the `java.sql.TimeStamp` values to the sensor, by using the `cal` calendar object. Therefore, in this example, the `_gmt` in the field name defined in `initializeDB` is a clue to the next person who edits the code that a GMT calendar should be used.

The sample code also makes it clear that the numeric values in the temperature field are in degrees Celsius. The JDBC standard states that, if no calendar is provided, time values are stored in the time zone of the JVM. Knowing what that is and knowing that it will never change is problematic in a cloud environment. See the blog post "Achieving consistent meanings for DATETIME values."

Always specify a calender when writing and reading time stamp values to and from the database, as shown in the following two code listings, in which the calendar object is created and then the calendar object is used.

```
privatestaticfinal Calendar
 cal=Calendar.getInstance(TimeZone.getTimeZone("GMT"));
```

```
conn = TSConnection.getConnection();
sql = "insert into thermos_vti values(?,?,?)";
PreparedStatement statement = conn.prepareStatement(sql);
statement.setInt(1, data.id);
statement.setTimestamp(2,
 new java.sql.Timestamp(data.tstmp.getTime()), cal);
statement.setFloat(3, data.celsius);
statement.executeUpdate();
```

Limit the data size

The demo app does not need readings from Refrigerator 23 from last year. The app does need to conserve space on the cloud server. To limit the size of the data stored in the time series container used for this application, define a window of data you care about, for example:

```
statements[i++] = "execute procedure TSContainerCreate( "
 + "    'thermometer_readings',  "
 + "    'rootdbs',  "
 + "    'thermometer',  "
 + "    0, "
 + "    0, "

 + "    '2014-01-01 00:00:00'::datetime year to second, "
 + "    'day', " // window interval
 + "    2,  "    // active window size
 + "    3, "     // dormant window size
 + "    null, "

 + "    1)";     // window control, As many as necessary existing
 //dormant partitions and older active partitions are destroyed.
```

Full details of this procedure can be found in the documentation for rolling window containers.

Define a virtual table to access time series data

Some operations are easier if you can access the time series as though it were a relational table. For those operations, define a virtual table, as shown in the following listing.

```
statements[i++] = "EXECUTE PROCEDURE TSCreateVirtualTab( "
 + " 'thermos_vti', 'thermometers', "
 + "'origin(2014-01-01 00:00:00), "
  + "calendar(onesec),container(thermometer_readings),threshold(0),irregular'"
 + ", 0)";
```

I use an irregular series in this application because regular time series are populated with null entries for every missing value between the origin and the most recent entry and with that method, space is wasted on entries for a few months or years of readings every second.

Learn more about using Informix TimeSeries in the publication *IBM Redbooks: Solving Business Problems with Informix TimeSeries*.

Sensor

The Sensor class emulates a sensor containing a thermometer that sends its reading to the server using standard `POST` messages. For this class, the `send()` method accepts any serializable object, as shown in the following listing.

```
privatevoid send(Serializable output)
{
 try
 {
  // send data to the servlet
  HttpURLConnection conn = getServletConnection("tswrite");
  conn.setDoInput(true);
  conn.setDoOutput(true);
  conn.setRequestMethod("POST");

  OutputStream outputStream = conn.getOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(outputStream);
  oos.writeObject(output);
  oos.flush();
  oos.close();

  int response = conn.getResponseCode();
  if (response != 200) // HttpStatus.OK
  {
   String message = conn.getResponseMessage();
   System.err.println(
   "Failure in sending value from sensor to database.\n"
   + message);
  }
 } catch (Exception ex)
 {
  ex.printStackTrace();
 }
}
```

The sensor readings are encapsulated within a Java object and the serialized object is transmitted over HTTP. As long as the receiver knows what class of object to expect, this is a general solution

and any number of fields can be sent as a unit. You do not have to change the methods for sending and receiving the data if the number or types of information being sent change.

The code that handles the passing of the sensor identifier and the statistics about the sensor readings uses the same methodology. The same code can work in a variety of implementations.

Monitor

The monitor is also a simple class. It is responsible for deciding when and how to alert the user. In this demo, the alert is simply changing the color of a button. Although it is easy to have the monitor send a text message, (for example, by connecting it to a messaging service), but that is beyond the scope of the TimeSeries service itself.

I use the Apache Commons Lang to create statistics on the data for a customer's device. This process consists of aggregating the most recent values sent by the device, as shown in the following listing.

```
SimpleRegression simpr = new SimpleRegression();
Mean mean = new Mean();
sql = "select tstamp_gmt, celsius "
 + "from thermos_vti "
 + "where "
 + "thermometer_id = ? "
 + "and "
 + "tstamp_gmt >= "
 + "(cast (? as datetime year to fraction(3)) "
 + "   - interval(4.5) second(2) to fraction(1))::datetime year to fraction(5)";
statement = conn.prepareStatement(sql);
statement.setInt(1, id);
statement.setTimestamp(2, new java.sql.Timestamp(time.getTime()), cal);
rs = statement.executeQuery();
while (rs.next())
{
 long millis = rs.getTimestamp(1).getTime();
 double celsius = rs.getDouble(2);
 simpr.addData(millis, celsius);
 mean.increment(celsius);
}
stats.recentMean = (float) mean.getResult();
stats.recentSlope = (float) simpr.getSlope();
```

The data analysis is very simple, but it could be made more sophisticated for another application. Every four seconds, the monitor passes the device identifier to a servlet running on Bluemix. The servlet creates the statistics and passes the results back to the monitor. The monitor contains a method that determines whether action should be taken.

It is possible to create a routine in the database to run the analysis. After this step is taken, it is a matter of programming preference whether to poll the statistics from outside the database or to created a scheduled task within the database server.

Conclusion

This application gives a simple demonstration of using Informix TimeSeries in an application that combines using an Internet of Things device with software services provided in Bluemix. Informix

TimeSeries makes it possible to predict when problems are likely to occur, before they actually happen. This set of steps is tied together with standard HTTP operations for exchanging data over the Internet.

RELATED TOPICS: Informix TimeSeries Java