Hi Professor,

As I said in class on Friday, I think it would be useful if we come up with a core set of methods that the iSENSE project will need to implement. I have come up with a set of methods that I think is both simple and flexible. The methods are not tied to a specific protocol (e.g., SOAP). I did this because I got the impression SOAP would just cause headaches for the class, and it may not be perfect for the type of system we need to implement.

I think, for this project, a simple custom protocol (like Matt described) would be more appropriate than using SOAP. The downside is that we would have to implement our own encryption (in addition to user authentication), since we won't have SOAP's built-in SSL capability to rely on. There are a number of (simple) ways to do this, and we can worry about that later (I have lots of experience with that). Also, although it's frowned upon, we could still run our protocol over standard HTTP/HTTPS ports, bypassing most schools' firewalls.

The set of methods I have come up with is also independent of which database we choose. Whether we use PostgreSQL or MySQL, the methods will work in essentially the same way. We wouldn't have to tie ourselves to one database initially. If additional functionality is needed (from PostgreSQL, for instance), we can add that later.

Instead of relying on a SOAP-based web service running on something like Apache Axis2, we would write our own server to handle user requests. This is very simple with Java. Users would be authenticated on a per-connection basis, so a user would open a connection and authenticate. From there on, that connection would have access to that user's data. If the authentication fails, the connection would be dropped. We could authenticate users against the Moodle database fairly simply.

Here is the prototype for the authentication method:

```java
/** Authenticates the user against the Moodle database. *
 * @param username Moodle username.
 * @param password Moodle password.
 * @return true if authentication succeeded, else false.
 */
boolean authenticateUser(String username, String password);
```

Next come the methods to manage the experiments. Experiments are associated with a single user in the database. It initially only has a title. Additional properties can be added if needed (more on that later). The “Experiment” type would likely just be a simple integer (as would the DataField and DataType types mentioned later).

```java
/** Creates a new experiment in the database. *
 * @param name Title of the experiment.
 * @return Experiment identifier on success, else null.
 */
Experiment createExperiment(String name);
```

```java
/** Retreives a list of experiment identifiers associated with user. *
 * @return Array of experiment identifiers on success, else null.
 */
Experiment[] getExperimentList();
```
/** Deletes the given experiment from the database. 
* 
* @param experiment Experiment identifier to delete.
* @return true if the experiment was deleted, else false.
* */
boolean deleteExperiment(Experiment experiment);

Experiments have data fields in which data can be stored. The data fields have a 
title and type associated with them.

/** Adds a new data field to the given experiment. 
* 
* @param experiment Experiment identifier being modified.
* @param name Title of the data field (e.g., "Temperature").
* @param type Type of data being stored (e.g., int).
* @return Data field identifier if it was added, else null.
* */
DataField addDataField(Experiment experiment, String name, DataType type);

/** Retreives list of an experiment's data fields. 
* 
* @param experiment Experiment whose field list we are retreiving.
* @return Array of data field identifiers if successful, else null.
* */
DataField[] getDataFieldList(Experiment experiment);

/** Retreives the data type of the given data field. 
* 
* @param experiment Experiment identifier.
* @param field Data field identifier.
* @return Data type if successful, else null.
* */
DataType getDataFieldType(Experiment experiment, DataField field);

/** Deletes the data field (and all associated data) from the experiment. 
* 
* @param experiment Experiment identifier.
* @param field Data field identifier.
* @return true if the data field was delete, else false.
* */
boolean deleteDataField(Experiment experiment, DataField field);

As I mentioned above, experiments can have a number of properties associated with 
them, in addition to their titles.

/** Adds a String property to the experiment. 
* 
* @param experiment Experiment identifier.
* @param property The name of the property to be added.
* @return true if the property was added, else false.
* */
boolean addExperimentProperty(Experiment experiment, String property);

/** Sets a given property in an experiment. 
* 
* @param experiment Experiment identifier.
* @param property Property name.
* @param value The new value of the property.
* @return true if the value was set, else false.
* */
boolean addExperimentProperty(Experiment experiment, String property);
boolean setExperimentProperty(Experiment experiment, String property, String value);

/** Deletes a given property (and its value) from the experiment. *
 * @param experiment Experiment identifier.
 * @param property Name of the property to delete.
 * @return true if the property was deleted, else false.
 */
boolean deleteExperimentProperty(Experiment experiment, String property);

Next come the methods for storing data in the experiments' data fields. The DataPoint class would have a field containing the actual data, along with a timestamp.

/** Stores an integer in the experiment in a given data field. *
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @param data The data to store.
 * @return true if the data was stored successfully, else false.
 */
boolean storeDataInt(Experiment experiment, DataField field, DataPoint<int> data);

/** Stores a string in the experiment in a given data field. *
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @param data The data to store.
 * @return true if the data was stored successfully, else false.
 */
boolean storeDataString(Experiment experiment, DataField field, DataPoint<String> data);

/** Stores a float in the experiment in a given data field. *
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @param data The data to store.
 * @return true if the data was stored successfully, else false.
 */
boolean storeDataFloat(Experiment experiment, DataField field, DataPoint<float> data);

/** Stores a boolean in the experiment in a given data field. *
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @param data The data to store.
 * @return true if the data was stored successfully, else false.
 */
boolean storeDataBoolean(Experiment experiment, DataField field, DataPoint<boolean> data);

Reading data points from the database is equally simple. The methods below return all the data from the requested data field.
/** Reads integer data from a given data field in an experiment.
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @return Array of data points if successful, else null.
 */
DataPoint<int>[] readDataInt(Experiment experiment, DataField field);

/** Reads string data from a given data field in an experiment.
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @return Array of data points if successful, else null.
 */
DataPoint<String>[] readDataString(Experiment experiment, DataField field);

/** Reads float data from a given data field in an experiment.
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @return Array of data points if successful, else null.
 */
DataPoint<float>[] readDataFloat(Experiment experiment, DataField field);

/** Reads boolean data from a given data field in an experiment.
 * @param experiment Experiment identifier.
 * @param field Data field identifier.
 * @return Array of data points if successful, else null.
 */
DataPoint<boolean>[] readDataBoolean(Experiment experiment, DataField field);

So users have experiments. Experiments have properties and data fields. Data fields contain the actual data. For example, I might create an experiment about temperature and humidity. I would first create the experiment with the title “Comparing Temperature and Humidity”. Then I would add two data fields of type “integer” to the experiment: one called “Temperature”, one called “Humidity”. Data could be then be stored in each data field and read back later by the visualizer.

I hope that all makes sense. I think it's a good starting point at least. Maybe you could get feedback in class on Monday?

-Will